Simulated Consultative Committee for Space Data Systems (CCSDS) Telemetry Generator (SCTGEN)

System User's Guide

September 1997

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland

SCTGEN
System User's Guide

September 1997

Prepared by:	**Quality Assured by:**
Stan Hilinski, ETS Lead Software Engineer **Integral Systems Inc.**	**J. Klein, Quality Engineer** **Unisys, Inc.**
Prepared by:	**Approved by:**
Khai Dolinh, ETS Software Engineer **Integral Systems Inc.**	**N. Speciale, Head** **Microelectronic Systems Branch** **Code 521, NASA/GSFC**
Reviewed by:	**Edited by:**
Chandru Mirchandani, ETS Project Lead **Lockheed Martin Space Mission Systems**	**L. Kane, Sr. Technical Writer** **NYMA, Inc.**

Goddard Space Flight Center
Greenbelt, Maryland

PREFACE

This document describes the software procedures for running SCTGEN. The user should note that the core software in SCTGEN is referred to as TIGER. All scripts and script errors in the main menu panel will refer to "Tiger Script". Tiger is an ongoing Code 521 in-house development effort.

This document will be under the configuration management of the Microelectronic Systems Branch Configuration Control Board (CCB). Changes to this document shall be made by Documentation Change Notice (DCN), reflected in text by change bars, or by complete revision.

Requests for copies of this document, along with questions or proposed changes, should be addressed to:

Technology Support Office
Goddard Space Flight Center
Greenbelt, Maryland 20771
(301)286-6034

CHANGE INFORMATION PAGE

List of Effective Pages	
Page Number	**Issue**
Title	**Original**
Signature Page	**Original**
iii through x	**Original**
1-1 through 1-5	**Original**
2-1 and 2-2	**Original**
3-1 through 3-104	**Original**
4-1 through 4-49	**Original**
5-1 through 5-18	**Original**
6-1 through 6-8	**Original**
A-1 and A-2	**Original**

Document History	
Issue	**Date**
Original	**September 1997**

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D)

FIGURES

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

SECTION I
INTRODUCTION

1.1 OVERVIEW

This document provides the procedures to use the Simulated Consultative Committee for Space Data Systems (CCSDS) Telemetry Generator (SCTGEN). The SCTGEN is being developed primarily as part of the Earth Observing System Data and Information System (EOSDIS) Test System (ETS) Project. SCTGEN's primary function is to generate telemetry data files in various formats according to user specifications. In conjunction with other ETS components, SCTGEN will be used to support EOS ground system integration, testing, verification, and validation.

SCTGEN is a software package with a graphical user interface that simulates CCSDS and non-CCSDS telemetry for both forward and return link data streams. When used as a test tool for the EOSDIS it will also simulate EOS Data and Operations System (EDOS)-generated data products, such as EDOS Data Units (EDU), Expedited Data Sets (EDS), and Production Data Sets (PDS). SCTGEN will also provide comprehensive error insertion capabilities. SCTGEN is an off-line tool used to generate test data, and as such will not present a schedule conflict with operational systems. However, to generate the required test data in time for testing, the user community will have to schedule adequate time to compose and generate the test data suite.

SCTGEN is being developed by Code 521, the Microelectronic Systems Branch (MSB), at Goddard Space Flight Center (GSFC). In addition to specific ETS simulation functions that are funded by the ETS Project, SCTGEN will also simulate other commonly used data formats, scenarios, and communication protocols as a generic data simulation tool. Generic efforts are funded by the MSB. All ETS Project needs and requirements pertaining to the same are specified in this document.

1.2 INTENDED AUDIENCE

This document is intended for:

a. ETS test and maintenance team who need to use the capabilities to generate scenarios to test the sub-systems that comprise ETS.

b. EOSDIS Independent Verification and Validation (IV&V) team and the ETS user community, who require a detailed set of guidelines to use the full capabilities of the system.

1.3 SYSTEM OVERVIEW

SCTGEN will be based on an architecture that provides a modular and flexible environment. This environment will support current data simulation requirements, and also allow future development of new data formats and a range of data scenarios. It will run on various UNIX platforms. High reusability is a basic design goal, with a perspective for both code and data reuse. The user interface will be based on graphical representation, and will be enhanced with intelligence to alleviate the cumbersome burdens involved in test data specification.

To support automated test operations, SCTGEN will provide a summary of expected results for each test data file generated. Expected results will be stored in easily readable formats so that ETS data verification tools can read them in for comparison with actual test results.

Figure 1-1 illustrates the conceptual ETS system architecture; shaded boxes represent SCTGEN's role of in the context of ETS.

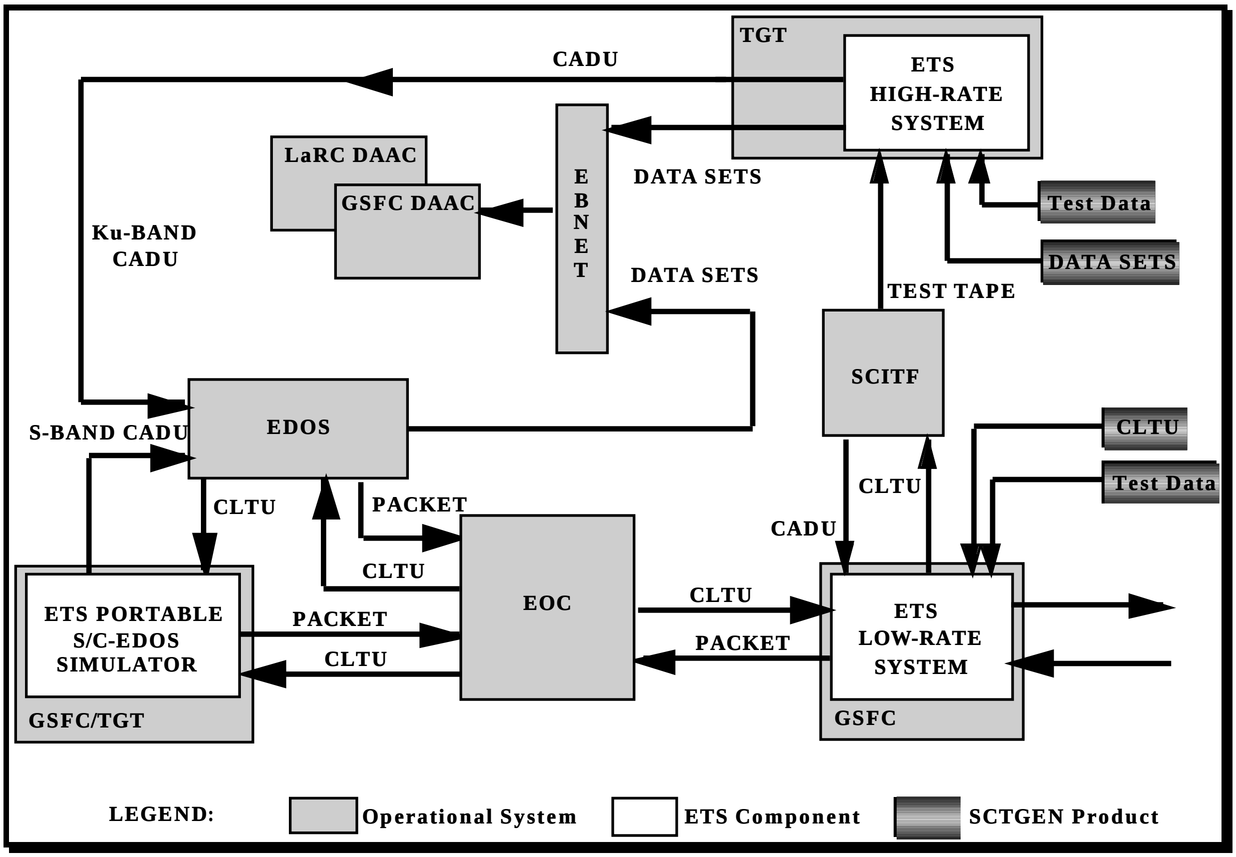

Figure 1-1. ETS System Architectural Overview

1.4 APPLICABLE DOCUMENTS

The following documents were used as references for development of system requirements. They further clarify, support, and define SCTGEN objectives, and the information in provided this document.

a. Earth Observing System Data and Information System (EOSDIS) Test System (ETS) Functional and Performance Requirements, 515-4FRD/0294, September 1995.

b. Earth Observing System Data and Information System (EOSDIS) Test System (ETS) Operations Concept, 515-3OCD/0194, May 1995.

c. Data Format and Control Book for EOS-AM Spacecraft, Interface Control Document (ICD) 106, Martin Marietta Corporation, Astro Space, April 1994.

d. Consultative Committee for Space Data Systems, Recommendations for Space Data System Standards: Advanced Orbiting Systems (AOS), Networks and Data Links, 701.0-B-2, November 1992.

e. Consultative Committee for Space Data Systems, Recommendations for Space Data System Standards: Telemetry Channel Coding, CCSDS 101.0-B-3, May 1992.

f. Consultative Committee for Space Data Systems, Recommendations for Space Data System Standards: Time Code Formats, CCSDS 301.0-B-2, April 1990.

g. Consultative Committee for Space Data Systems, Recommendations for Space Data System Standards: Packet Telemetry, CCSDS 102.0-B-3, November 1992.

h. EOS Data and Operations System (EDOS) - EOSDIS Backbone Network (EBnet) Interface Control Document, 510-ICD-EDOS/EBnet, August 1995.

i. ETS-EBnet Interface Control Document (TBS).

j. Tracking and Data Relay Satellite System (TDRSS) Ground Terminal (TGT) - EDOS Interface Control Document (TBS).

k. EDOS-Distributed Active Archive Center (DAAC) Interface Control Document (TBS).

l. EDOS Functional and Performance Specification, 560-EDOS-0202.0004, NASA/GSFC, December 18, 1992.

m. EDOS External Interface Control Document Data Format Control Document, July 1995.

n. High-Rate Data Test Equipment for EOS-AM Spacecraft Requirements Document, GSFC, NAS5-32500, Martin Marietta, March 1994.

1.5 DOCUMENT ORGANIZATION

This document is organized as follows:

Section 1 provides introduction.

Section 2 defines external interfaces as they pertain to SCTGEN.

Section 3 describes the SCTGEN Graphical User pertaining to the simulation needs of ETS.

Section 4 provides the reference documentation needed to understand the scripts.

Section 5 contains the complete user documentation for the scripting language.

Section 6 contains examples of sample scripts and test data.

1.6 TERMINOLOGY

1.6.1 DEFINITIONS

Data Format: Refers to data unit structure. Includes, but is not limited to, data unit size, description of fields within the data unit and any constraints therein, and format identification.

Data Layer: Refers to level of encapsulation of the source data. Packet data layer implies that source data is encapsulated in CCSDS or non-CCSDS packets. Frame or Virtual Channel Data Unit (VCDU) data layer implies that source data is encapsulated within frames or VCDUs, etc.

Data Scenario: Refers to design of test data to be generated. Includes the order in which data units are multiplexed within each level, the order in which data units are multiplexed at the next level of encapsulation, etc. For example, the order of packets (as distinguished by Application Identifiers [APID]) within each Virtual Channel Identifier (VCID), and the frequency with which these VCIDs are encapsulated into CADUs and multiplexed to form a CADU test data product.

Data Segment: Refers to a collection of time-ordered packets. A data stream may have many collections of time-ordered packets within the collection, but out of time order from collection to collection (for example, a real-time downlink and a playback dump).

Data Stream: Refers to a serial stream of binary data contained in a test data product.

Data Unit: Refers to level of encapsulation of actual user data. For example, a packet data unit consists of a header, source data, and trailer; a frame data unit consists of a header, data unit zone, and trailer. The frame data unit zone may contain packet data units, or pure source data.

Specifications: Refers to user-defined parameters for the generation of test data. Includes data format, data scenario, level of encapsulation, special instructions for data manipulation (what, where, and how many), error insertion, timecode deviation, gap insertion, and size of final product.

Test Data: Refers to final product generated by SCTGEN. This could be a concatenated stream of bit stream data, packet data units, frame data units, block data units, etc., stored in file format, or as a binary stream on tape.

Data Patterns: Refers to data within the application data zone of a packet. This data can be any pattern, either defined by the user, or read in from a file.

Packet Stream: Refers to concatenated serial stream of multiplexed packets from a single virtual channel.

Bit Stream: Refers to a single stream of data bits encapsulated in frames.

Frame Stream: Refers to a concatenated serial stream of multiplexed frames that comprise a single data stream.

1.6.2 ACRONYMS

APID	Application Identifier
CADU	Channel Access Data Unit
CCB	Configuration Control Board
CCSDS	Consultative Committee for Space Data Systems
CDS	CCSDS Day Segmented
CLCW	Command Link Control Word
CLTU	Command Link Transmission Unit
CODA	Customer Operations Data Accounting
CVCDU	Coded Virtual Channel Data Unit
DAAC	Distributed Active Archive Center
DCN	Documentation Change Notice
DFCD	Data Format Control Document
DSN	Deep Space Network
EBnet	EOSDIS Backbone Network
EDOS	EOS Data and Operations System
EDS	Expedited Data Set
EDS	Expedited Data Sets
EDU	EDOS Data Unit
EOC	EOS Operations Center
EOS	Earth Observing System
EOSDIS	EOS Data and Information System
ESDIS	Earth Science Data and Information System
ETS	EOSDIS Test System
GN	Ground Network
GSFC	Goddard Space Flight Center
ICD	Interface Control Document
IV&V	Independent Verification and Validation
LaRC	Langley Research Center
LZP	Level Zero Processing
M_PDU	Multiplexed Protocol Data Unit
NCC	Network Control Center
OMD	Operations Management Data
PDS	Production Data Set
PDU	Protocol Data Unit
PSS	Portable Spacecraft Simulator
RDF	Rate-Buffered Data File
SCID	Spacecraft Identifier
SCITF	Spacecraft Integration and Test Facility
SDU	Service Data Unit
SN	Space Network
STGEN	Simulated Telemetry Generation
TCP/IP	Transmission Control Protocol/Internet Protocol
TDRSS	Tracking and Data Relay Satellite System
TGT	TDRSS Ground Terminal
TPCE	Telemetry Processing Control Environment
TPGEN	Test Pattern Generator
TSS	TDRSS Service Session
VCID	Virtual Channel Identifier
VLSI	Very Large Scale Integration
WOTS	Wallops Orbital Tracking Station
WSGT	White Sands Ground Terminal

SECTION 2
INSTALLATION

2.1 INTRODUCTION

This section provides information on SCTGEN installation as it pertains to the ETS project.

2.1.1 INSTALLATION REQUIREMENTS

a. **Load SCTGEN on to the work-station platforms in pre-created directories.**

b. **Ensure that there are Ethernet taps off EBnet, one for the HRS-CDS, one for the LRS-CDS, and one for the TRS.**

2.1.2 INSTALLATION PROCEDURES

a. **Verify that the system Internet Protocol (IP) addresses are correct. If the system is not on a local network, check with the System Administrator to ensure that the addresses are acceptable.**

b. **Connect external Ethernet ports to the TRS and HRS, LRS workstations.**

c. **Connect internal FDDI port to the VHS and HRS workstation (optional).**

d. **Bring up the Menu Controller on the workstation.**

e. **Verify that the hard disks are turned on.**

f. **Turn on power to the chassis.**

g. **Verify boot process chassis.**

2.1.3 DIRECTORY CONTENTS

The directory contents of SCTGEN are described in the following paragraphs. The complete contents of the directories are described in the SCTGEN Delivery Document.

2.1.3.1 SCTGEN Graphical User Interface Release 1.01

The source and executables are contained in the listing under the directory titled ‘sctgen1.01. The files and directories in /sctgen1.01 are described as follows:

bin<dir>
data<dir>
doc<dir>
script<dir>
source<dir>

2.1.3.2 SCTGEN Software Release 1.0

The source and executables are contained in the listing under the directory titled 'Tiger'. The files and directories in /Tiger are described as follows:

/Tiger:

README
login
revision.doc
bin<dir>
    estp
    sctgen
h<dir>
lib<dir>
scripts<dir>
src<dir>
work<dir>
docs<dir>
    reference<dir>
    user<dir>

SECTION 3
SCTGEN GRAPHICAL USER INTERFACE

3.1 TOP-LEVEL MENU DESCRIPTIONS

The SCTGEN GUI will run on the CDS, and has been developed using TCL/TK tools. The Menu Controller running on the work-station will have an icon that will start up the main panel of the SCTGEN GUI. In the absence of the Menu Controller, the GUI may be started up by typing in 'sctgui'. The main panel or window will allow the user to select, for the different ETS test data products such as CADU, PDS, EDS, TC and packet products, the following SCTGEN options:

1. Create a new scenario
2. Load a previously created scenario
3. Clear a previously loaded scenario
4. Generate a script from a scenario
5. Generate a data file from a script
6. Create and insert error, events and tasks for a scenario

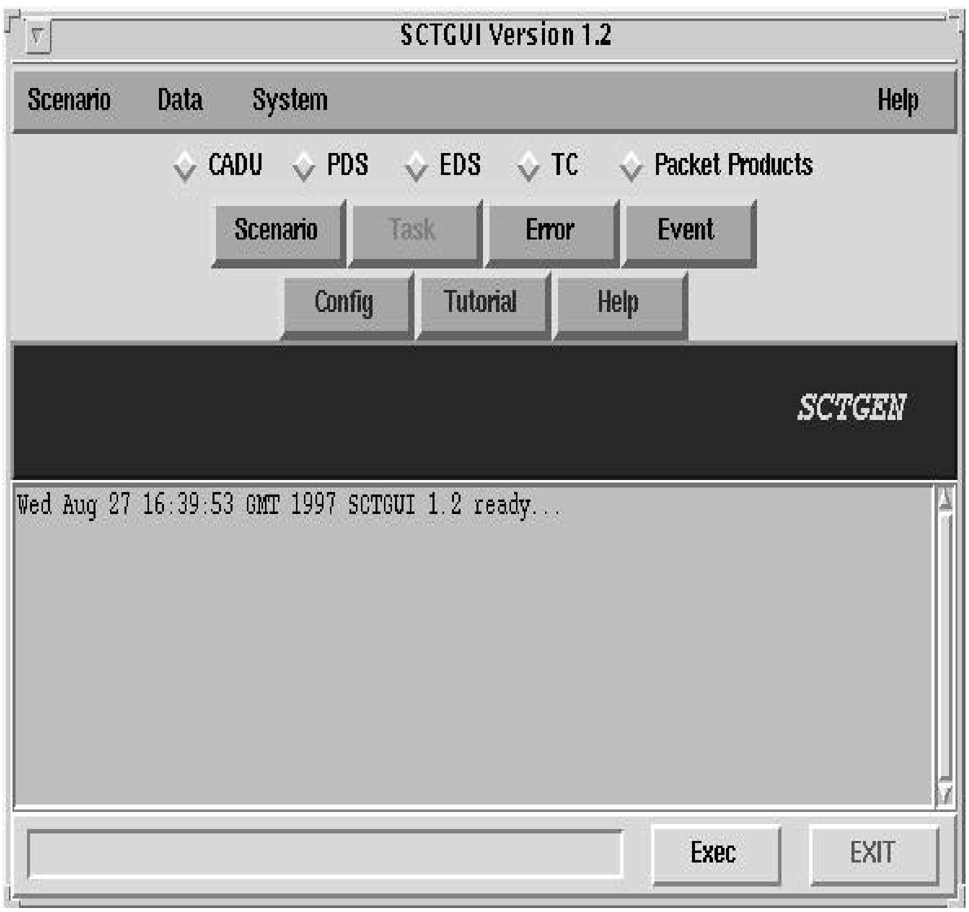

Figure 3-1. SCTGEN Main Menu Panel

However, some of these options are available only after the test data scenario has been designed and formatted as per the user's needs. Figure 3-1 shows the first menu of the SCTGEN system and the following paragraphs will describe the menus for the different test data development sequences.
Note: the user must exit the SCTGEN main menu before exiting from the UNIX window that started up the SCTGEN Graphical User Interface.

3.2 RETURN LINK DATA SCENARIO DEVELOPMENT

The first step is to develop a scenario for data generation. The example illustrated in Figure 3-2, shows the input for the generation of a return link CADU stream. The CADU button is first selected followed by the new Scenario option from the scenario pull-down menu.

The information that is input at this stage is the Spacecraft Identifier (SCID), VCID, and APID. Once these fields are selected the depression of the view button will pop up the next menu panel, Figure 3-3. Figure 3-3 shows the graphical description of the Data Scenario defined in Figure 3-2.

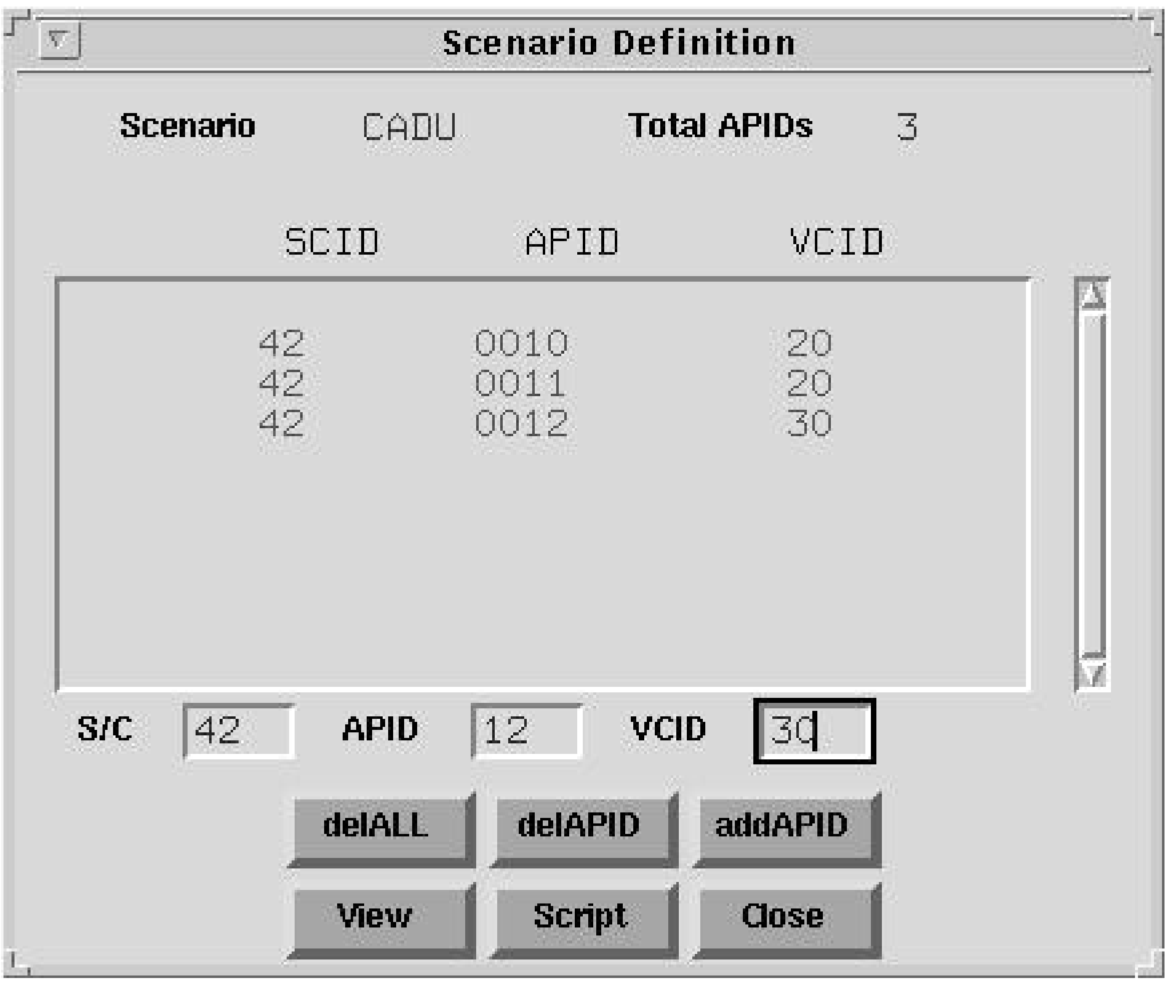

Figure 3-2. Data Scenario Menu Panel

From this point on each of the icons shown on the CADU stream menu open up to more detailed windows, for example: sc42out, will open the Output Definition Menu Panel shown in Figure 3-4, which will allow the user to define the number of frames, the format of the File being created, and the Device that is to be generated as a product.

For Output type, the user should select File or Sim. The other two options are not applicable with this version of the software.

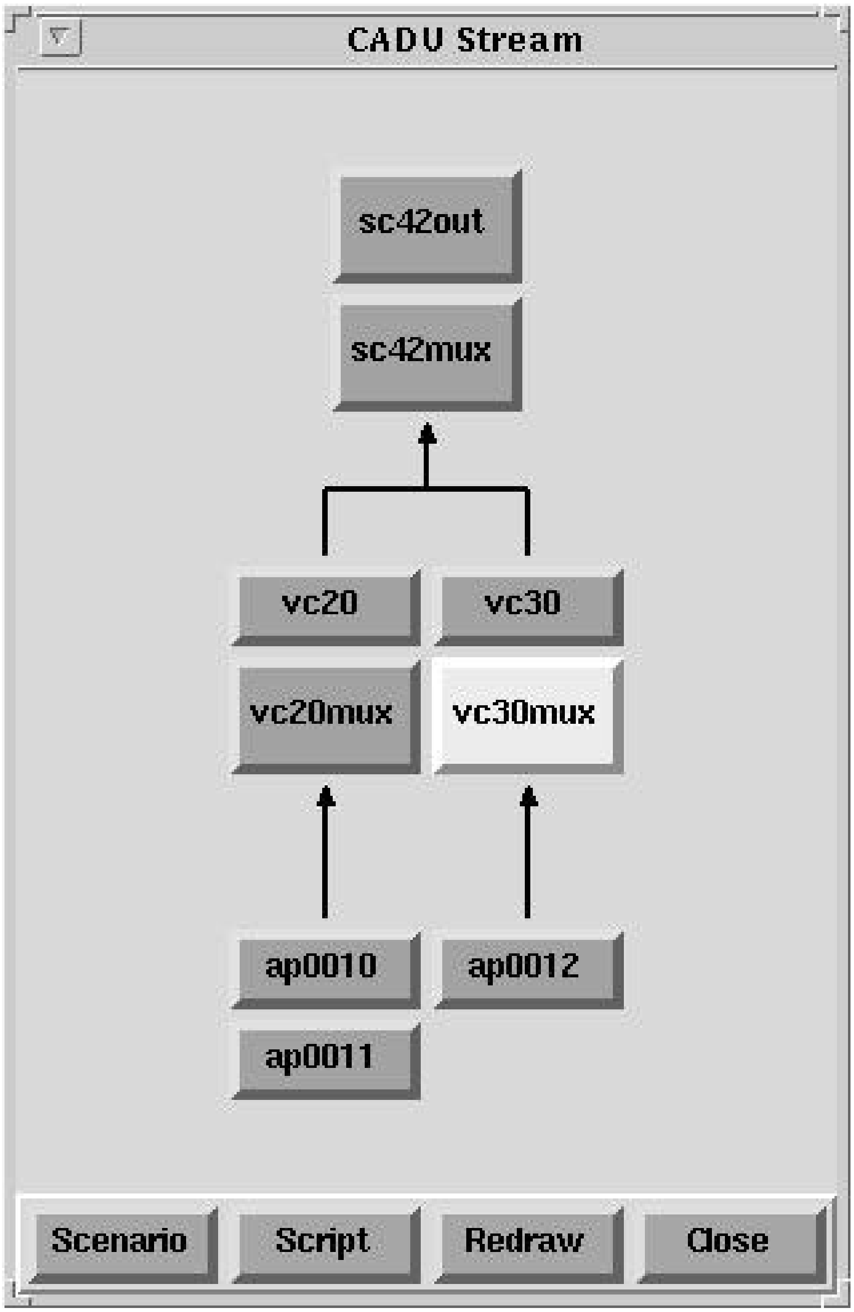

Figure 3-3. CADU Data Scenario Menu Panel

When ‘File’ is selected, this means that the output that is created is a plain file. ‘Sim’ is selected when creating test data to be loaded on to the ETS Simulator Card for subsequent output as a CADU data stream. The Simulator card DRAM is loaded with the base file and the update file refreshes the base file with updates via the on-board FIFO. This means that two files are created, the base file, ‘filename.b’, and the update file, namely filename.u’*. The number of units in this case defines the number of frames the user wants in this test data file. The File format defines whether the output file has a header and trailer or is just a plain stream of frames. Again, in the case of ETS the ‘File”option for the return link data is in a plain format. The File size is applicable when the test data to be created as a number of files, the size of each file can be selected.

Note: In the case of the ‘SIM’ format, the maximum number of records per side is 2048 for EOS-AM 1024-byte frames. In all cases this number has to be divisible by 4. Do not use the 100 units default, as a minimum of 4096 is needed if the whole 4M of memory is to be filled.

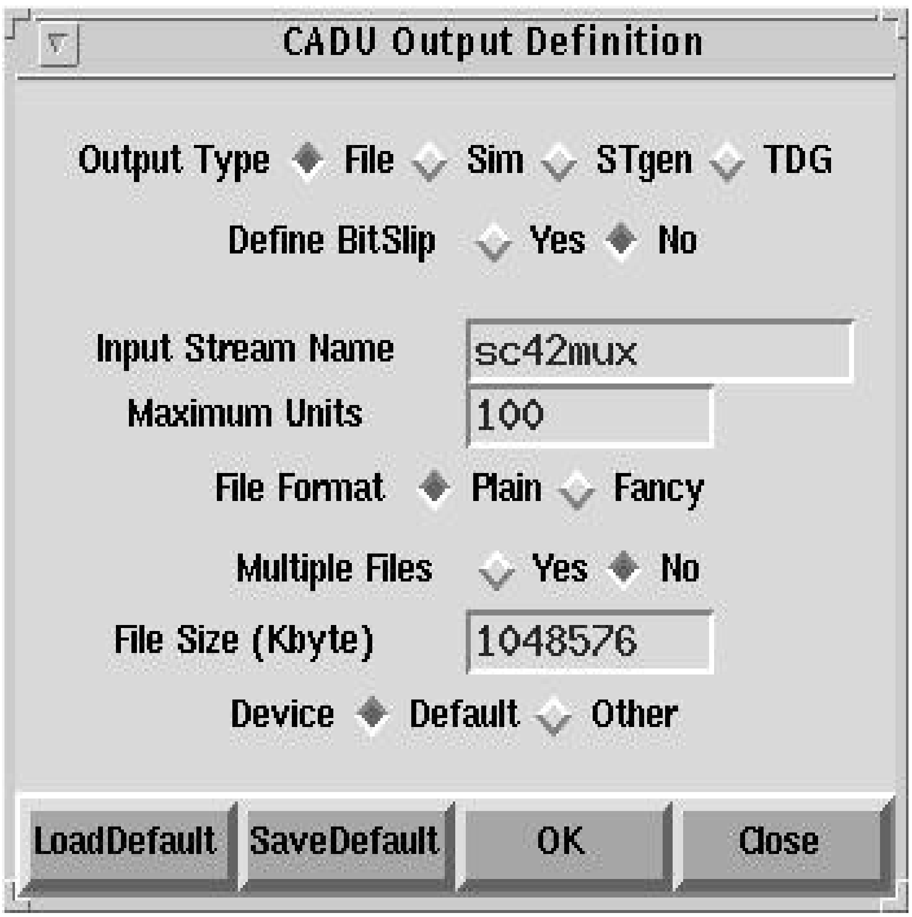

Figure 3-4. Output Definition Menu Panel

To define a file name Device should be selected by depressing the ‘other’ button. This will bring up the next menu panel, Figure 3-5 Device Definition. Here the Access mode is defined to be ‘write’, and the file name and extension are user selectable. The Device type implements either the creation of the test data file or the pseudo-generation of the data file, i.e., if the user wants to see the effects of the multiplexing and error insertion strategy without actually generating the data, the ‘null’ option together with the option to create an expected results file. allows only the Expected

Results file to be created, with an extension of 'filename.er'. Note, the option to create an Expected Results File has to be specifically entered. The menu panel that enables the creation of the expected results file will be described later on in this scenario.

Once the Device is defined, the OK button must be depressed to save the input at this level. This will take the user back to the Output Definition Menu, where once again the OK button has to depressed to save the input at that level. Depressing the close button will nullify the input entered by the user. Closing this menu will take the user back to the graphical menu where the next button, 'sc42mux' in the hierarchchy is depressed. This button will open up the Frame Multiplexer for sc42 Menu Panel, as shown in Figure 3-6.

This menu panel will allow the user to select the interleaving strategy for the frames that comprise the CADU stream by VCID number. There are two ways that the strategy can be defined, one by specifying the Frame content by range, i.e.,, in this option the data source is selected by depressing either the 'idle', or one of the available 'vcxx' buttons.

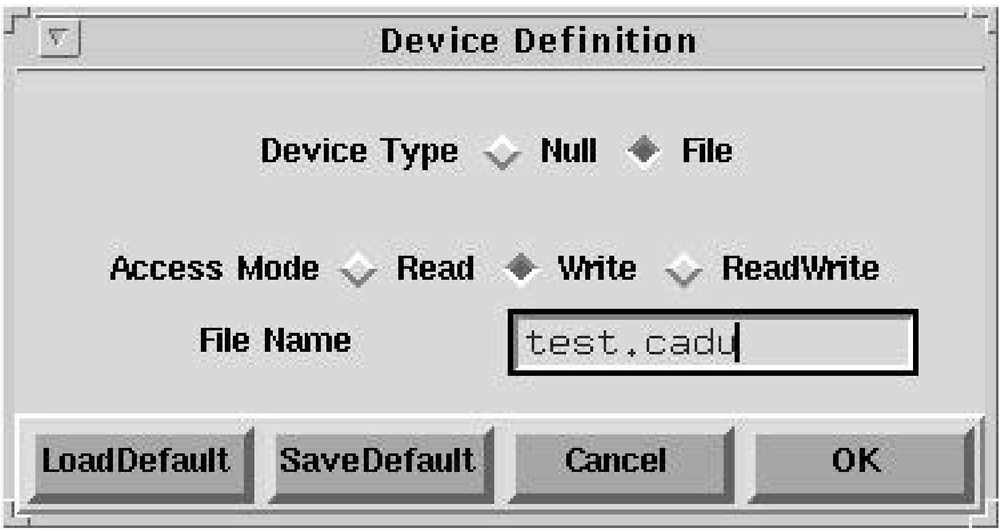

Figure 3-5. Device Definition Menu Panel

The idle frame may be created by any one of the available vc sources by selecting the vc source and depressing the 'SetIDLE' button. The start frame count and the stop frame count is then entered for the selected VCID or Idle frame. The next 'vcxx' is then selected and the start and stop frame count is entered. The other option is to use the 'set pattern option'. Here, once again the 'vcxx' or idle is selected and the start, the number of frames in an instance, the total number of frames in the pattern and the number of times the pattern is to be repeated is input. For example, 'vc2' starts at CADU count number 2, i.e., 'Start FRAME = 2'; appears once, i.e., 'No. FRAME of Data = 1'; is repeated once every two frames, i.e., /Total FRAME in Pattern = 2'; and this pattern is repeated forever, (until the selected number of CADUs are created), i.e., 'Repeat Count for Pattern = 0'*. An option to input the multiplexing strategy via a file, invoked by depressing the 'MUXtool' button will be implemented in the future (TBD). In either of the two options once the multiplexing strategy is defined the 'Set Range' or 'SetPattern' button has to be depressed before the menu panel is closed to save the input. Once the multiplexing strategy is entered, the View button, elongates the Menu Panel, showing the user the composition of the 'scxxmux' stream by VCID and the percentage of each 'vcxx' within the 'scxxmux' stream. The CLOSE option, closes the menu panel and takes the user back to the graphical display of the CADU Stream scenario.

The next step in the hierarchy is the Frame Definition. The number of VCIDs defined provides the number of Frame definition options available to the user. Suppose we have two VCIDs defined, thus there are two Frame Definition Menu Panels that have to be brought up and values introduced. On depressing the 'vc1' button, the Frame Definition Menu shown in Figure 3-7 is displayed.

The first two values on the menu panel are already defined from the initial input at the top level, i.e., Figure 3-2. For 'Service', if we select Path service, this means that the CADUs will contain Packets identified by APIDs. By selecting 'Path', the Service Definition Menu Panel is displayed. In this version of SCTGEN for ETS, no values are entered in this menu.

Once the Service is defined, the OK button will save the entries, and the CLOSE button will return the user to the Frame Definition menu.

Note: (Start, Date, Span) in this case Figure 3-6 refers to the first three entries, i.e.,, Start=Start Frame, Data=No. Frame of Data, Span = total frame in pattern to be replaced.

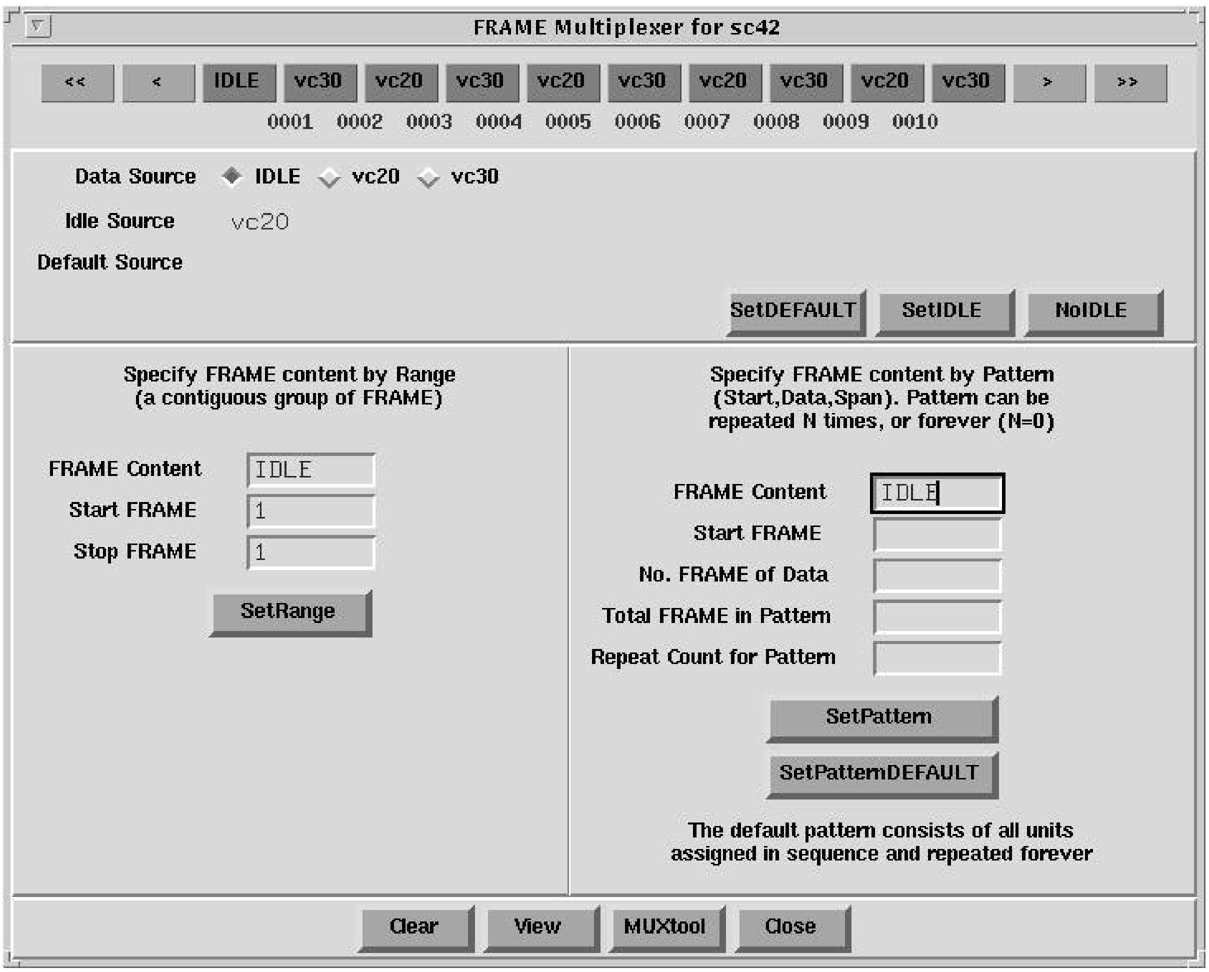

Figure 3-6. Frame Multiplexer Menu Panel

The next three entries in Figure 3-7 are user selected values for Frame length; the maximum number of Frames to be created from VCID being defined , i.e., 'vc1' in this definition; and the Fill pattern to be used in the VCID stream for Idle or Padded frames. For EOS-AM, the frame length is 1024 bytes. The next entry defines the flags in the frame header, to be Realtime or Playback, and the Event button allows the user to define an Event. The option to select a Frame Sync, displays the SYNC Definition Menu Panel, shown in Figure 3-8, that accepts that Sync Length and the Sync Pattern. Depressing the OK button saves the input and CLOSE returns the user to Figure 3-7.

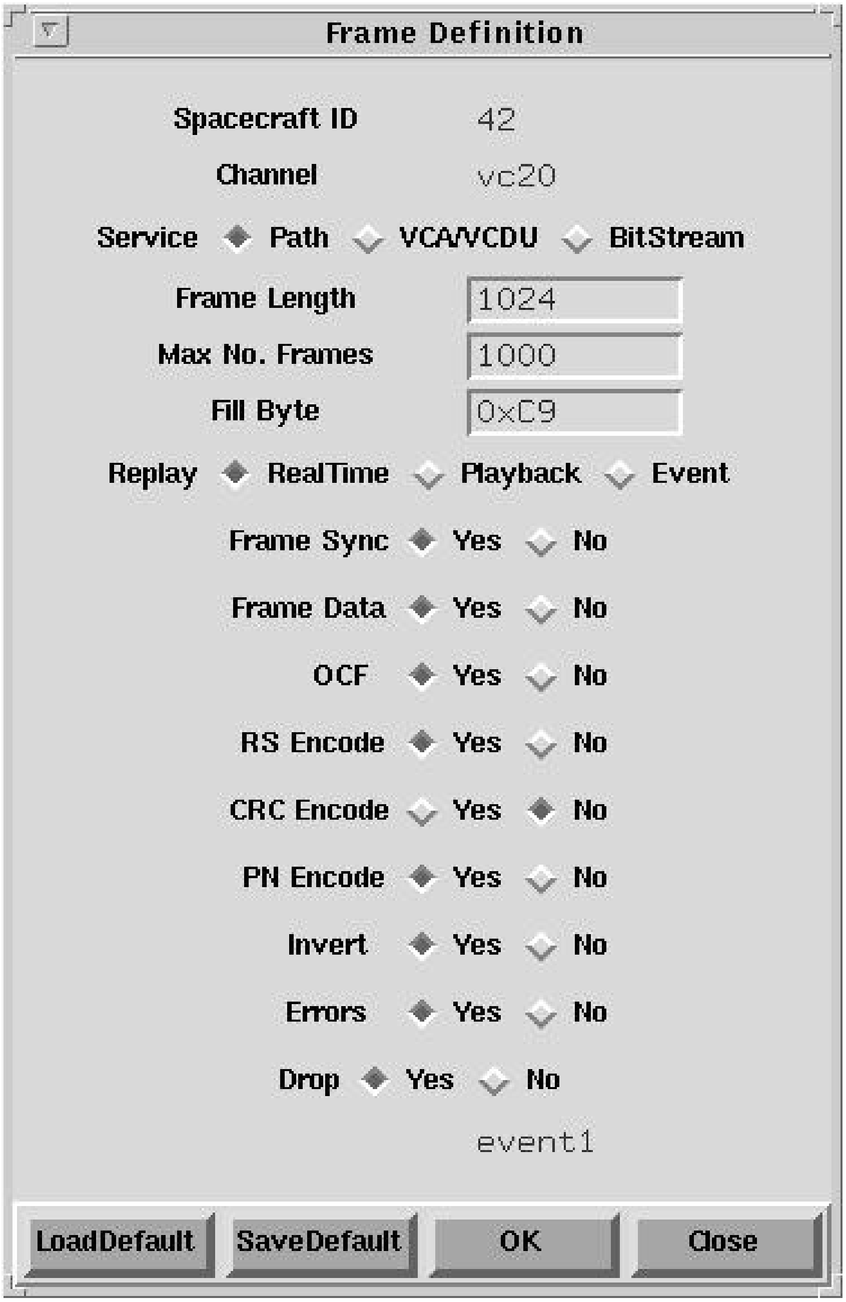

Figure 3-7. Frame Definition Menu Panel

When Path Service has been selected, the Frame Data option still has to be selected to define the packet mux that will feed this Virtual Channel stream, as shown in Figure 3-9. For cases other than Path Service, the Frame Data Definition Menu Panel, shown in Figure 3-10, will be displayed, to define the Data Region within the Frame.

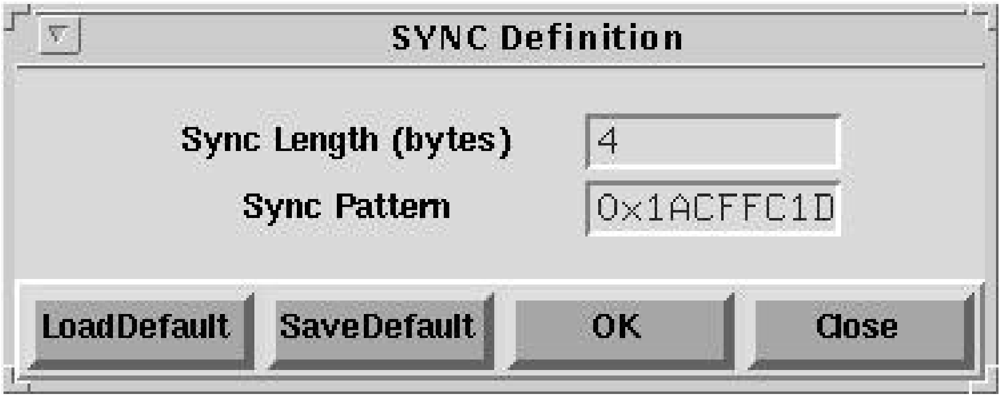

Figure 3-8. SYNC Definition Menu Panel

Figure 3-9 enables the user to optionally select the creation of the Expected Results File, which will by default have the name of the data scenario with a different extension, i.e., 'scenariofilename.er'.

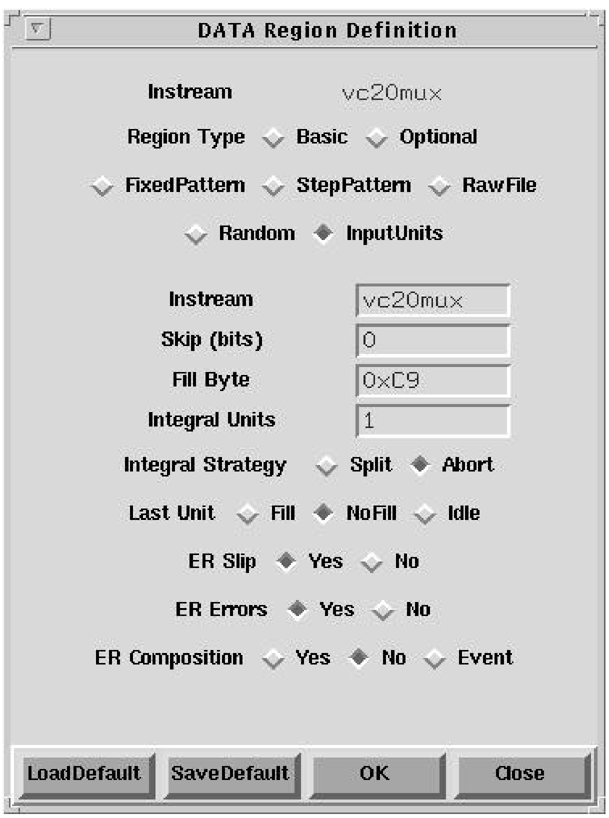

Figure 3-9. Frame Data Definition Menu Panel

The Frame Data can be a 'Fixed' pattern, 'Random' pattern, 'Step' pattern, or even read from a 'File'. When the 'File' option is selected, the name and path of the file have to be entered. In all the other cases the value has to be entered. OK and subsequently CLOSE, will save the input and return the user to Figure 3-7.

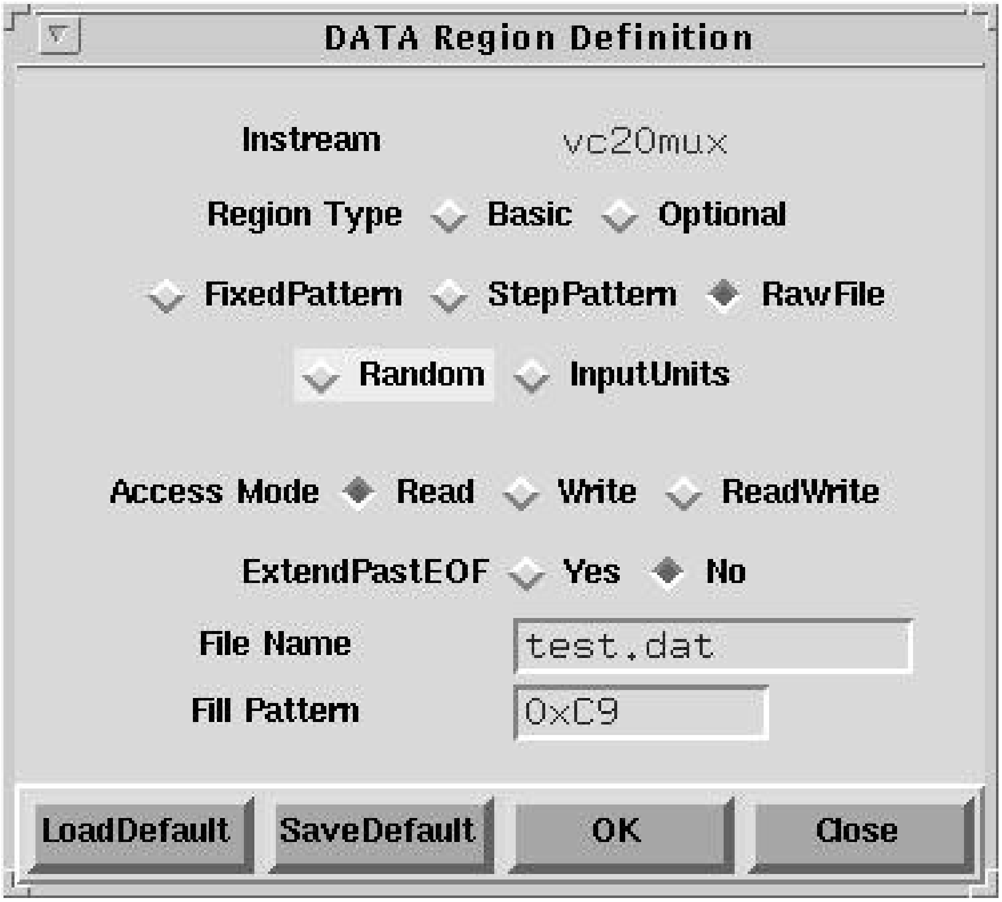

Figure 3-10. Frame DATA Definition Menu Panels

For Real-Time data the user may want to insert an OCF field in the CADU. The 'Yes' option to the OCF button will display the OCF Definition Menu Panel, shown in Figure 3-11. The length of the OCF and the content may be defined in this menu. The content may optionally be read from a file, in which case the full path name of the file has to be entered. OK and subsequently CLOSE, will save the input and return the user to Figure 3-7.

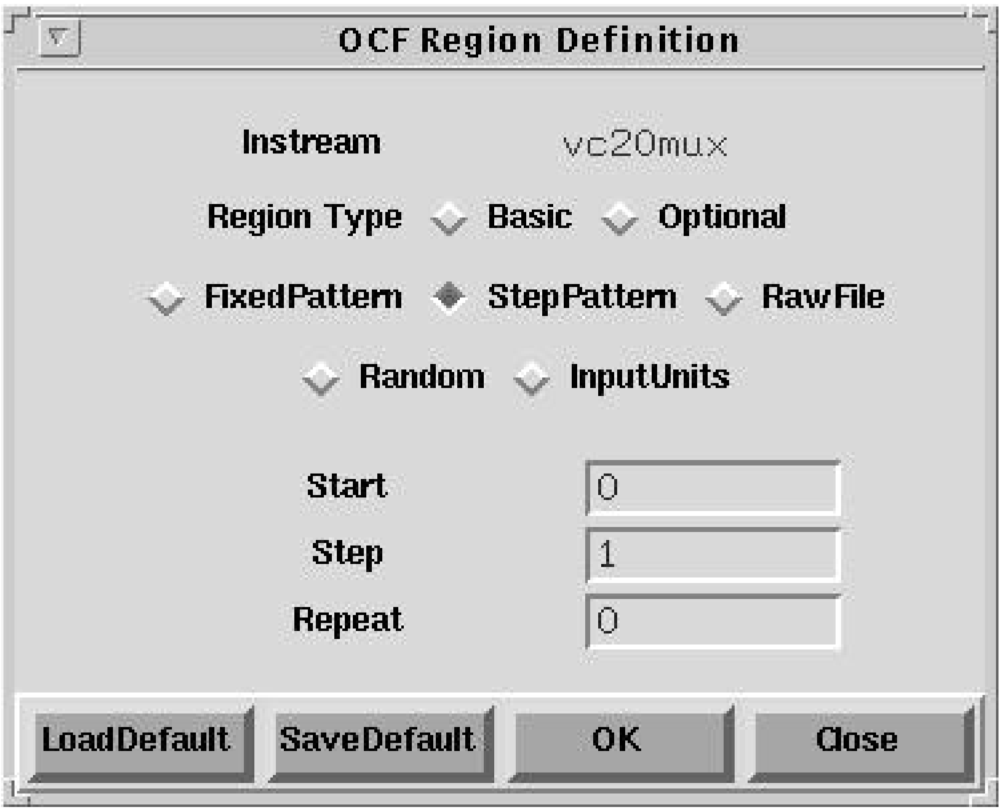

Figure 3-11. OCF Definition Menu Panel

The RS encode option when selected will display the RS Definition Menu Panel, shown in Figure 3-12. An RS interleave level from 1 to 16 may be selected. The RS Dual option and the RS Empty option may be selected or deselected independently. When the RS Empty option is selected, this means that SCTGEN will insert zeros in place of the Reed-Solomon check symbols at the end of each frame. OK and subsequently CLOSE, saves the input and returns the user to Figure 3-7.

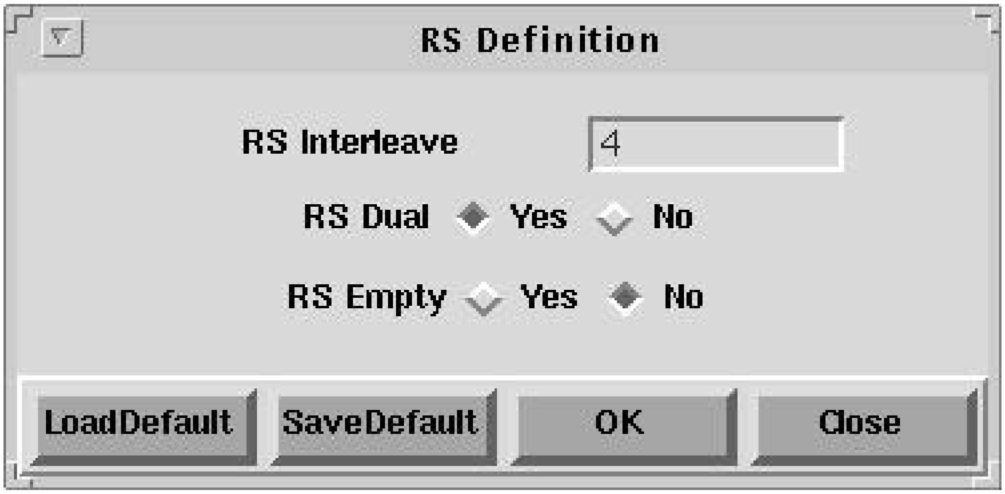

Figure 3-12. RS Definition Menu Panel

The CRC, PN and Invert options are not used for EOS-AM and are not available at this time. Finally the user has the capability to insert errors in the Frame at CADU Frame specific fields. Namely, the Frame Header fields, and the Frame Data Region. Selecting the 'Yes' option on the next entry in the Frame Definition, will display the Error Listing Menu Panel, shown in Figure 3-13.

For all VCIDs, any number of errors may be introduced. In the Error Listing Menu, the AddError is depressed to display the default Error Definition menu panel, shown in Figure 3-14, which comes up for the 'Flip' error type. In this menu, the user can label the type of error; select the method by which the error is to be inserted, i.e., by flipping bits, or setting bits to specific values, or by completely dropping frames identified by the VCID index or count.

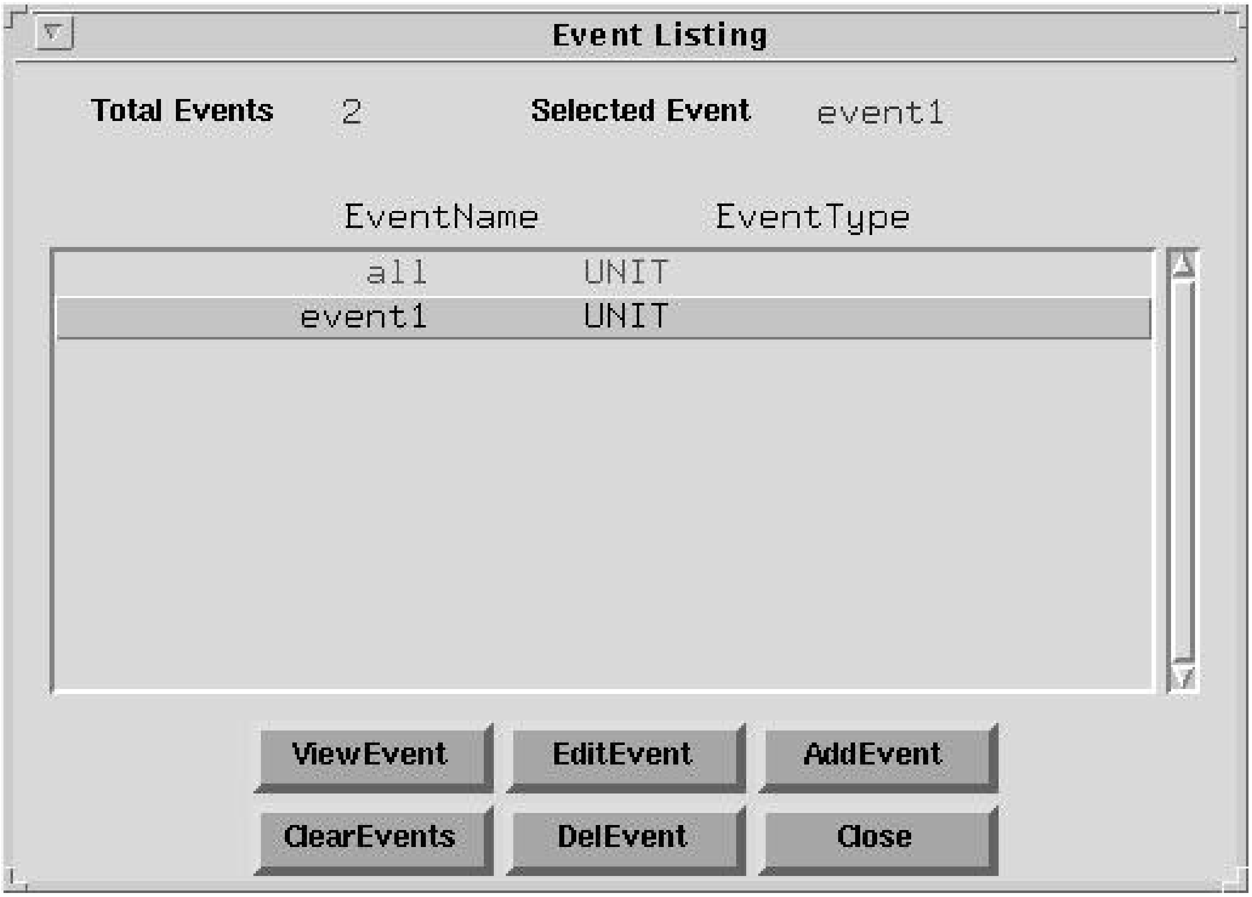

Figure 3-13. Error Listing Menu Panel

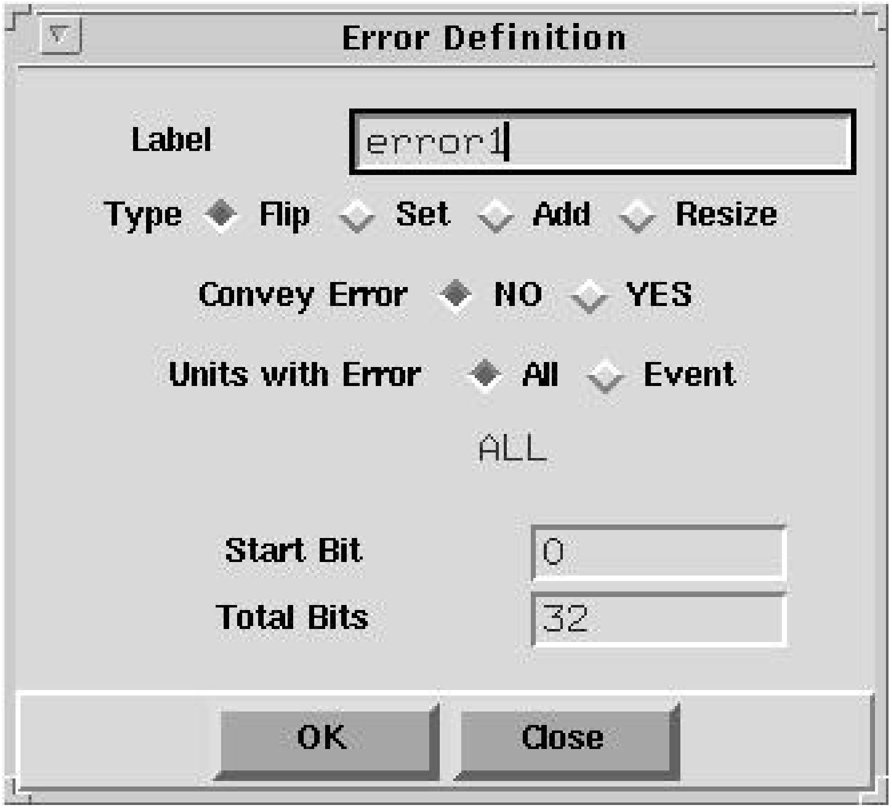

Figure 3-14. Flip Error Definition Menu Panel

When bits are to be flipped, Figure 3-14, the exact location of the bits within the frame, and the number of bits to be flipped are specified. The user can also specify whether this error event is to be implemented on all the Frames from this VCID stream or certain frames or ranges of frames as specified by their index in the stream. If all the frames are selected to have the error then the 'All' option is selected. If specific units are ear-marked for error insertion, then the 'Event' option i.e.,, the instance when a change is to be implemented, is selected which displays the Event Listing Menu Panel, shown in Figure 3-15.

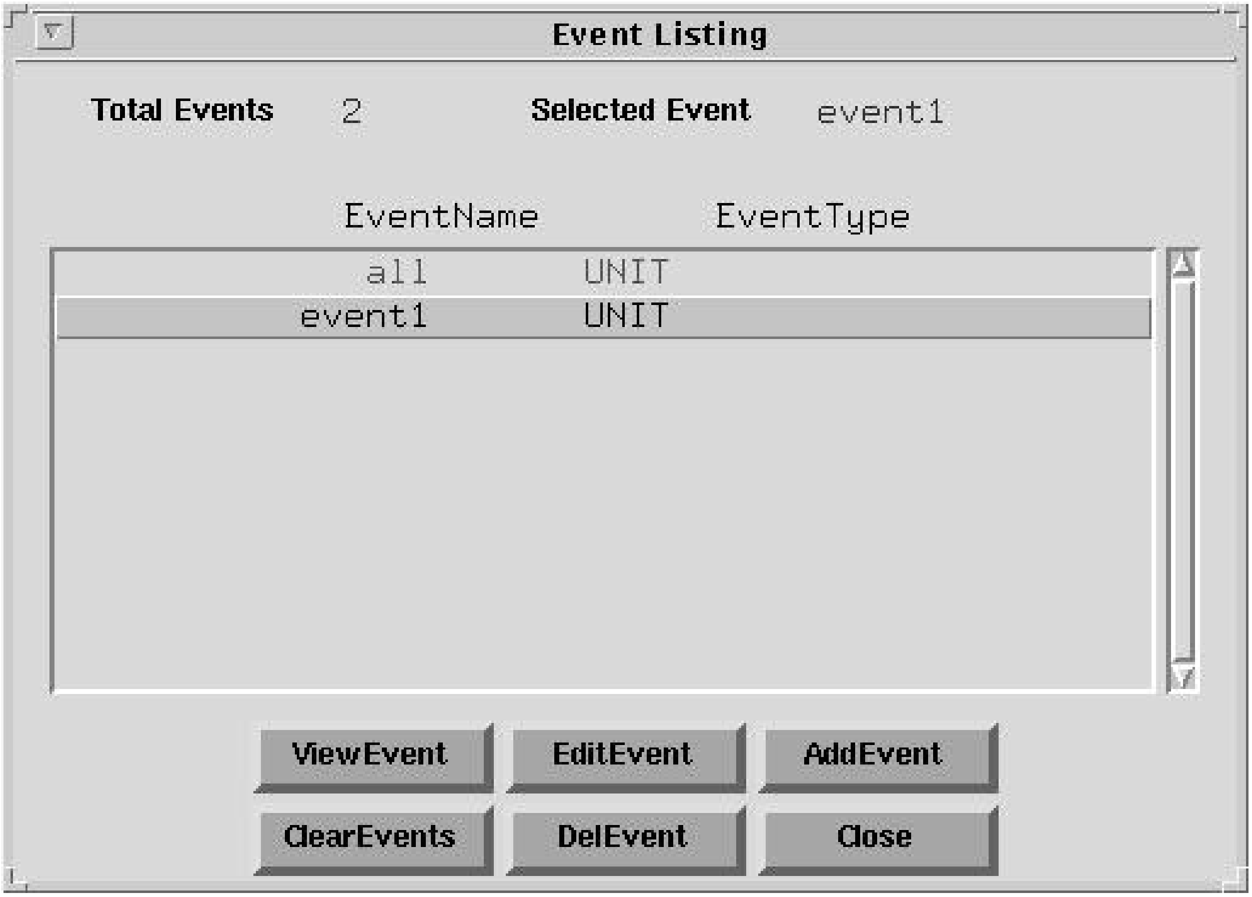

Figure 3-15. Event Listing Menu Panel

The AddEvent option displays the Event Definition Menu Panels, shown in Figure 3-16a, that allow the user to select the frames for error insertion. The Event name is user selectable and the occurrence of the event can be either a value or a unit. The Event Spec defines whether a certain range of units or frames specified by their unit number in the stream, or a recurrent pattern is specified based on the start frame number; how many contiguous frames from this frame stream; how many frames are to be skipped; and how many times this pattern is to be repeated, Figure 3-16. The section on the script descriptions will provide examples of both options. The OK and CLOSE options will save the selected input and return the user to the Error Definition Menu Panel.

Similarly when bits are set to certain values, the user can select the label, occurrence, location and quantity of the bits and the value to be set, Figure 3-17. In dropping Frames, the user can select frames by their index or unit number in the VCID stream. The Convey function is optionally set to convey the error up to the RS encoding process. If this is not set, the RS encoding process will correct the errors inserted in the Frame, and the created test data will not contain the Frame level errors. The other buttons shown in Figure 3-13 are self-explanatory. The View* Error button lists all the Errors; the Edit Error Button allows changes to selected errors as labeled; the DelError button allows the user to selectively delete an error entry, and finally the Clear Error button clears out all the listed errors.

Note: The “all” event listed on the Event Listing menu panel is a system event. This event appears in the script, but it is ignored unless the “all” event is defined for that particular script.

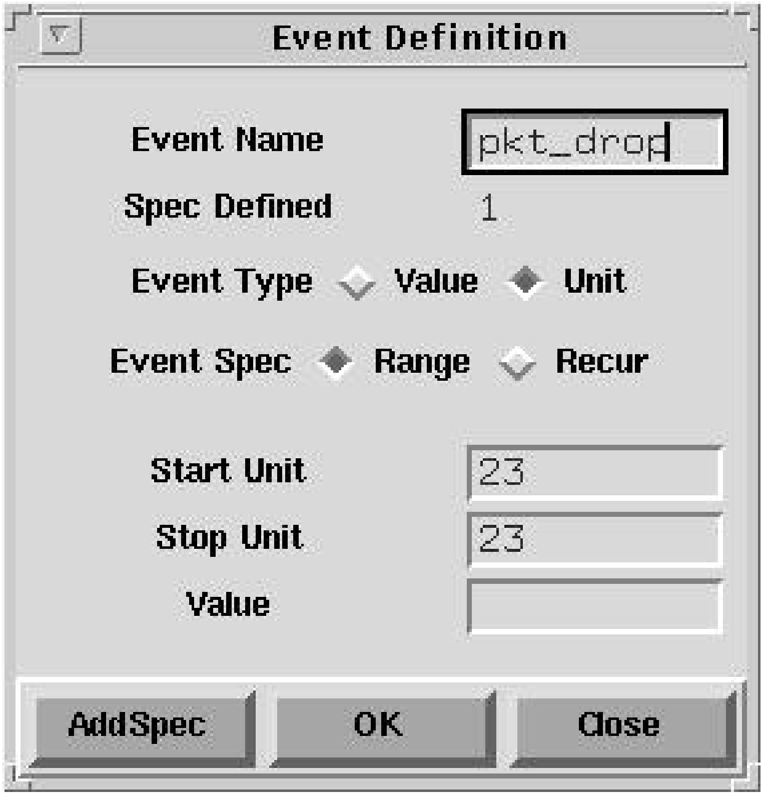

Figure 3-16a. Event Definition Menu Panels

In this version of the SCTGEN GUI, the user can not use the option to view or exit previouly created events using the Event Listing menu panel. One option is to view the SCRIPT using the script window and manually edit the script. The other option is to delete the selected event and re-define the event.

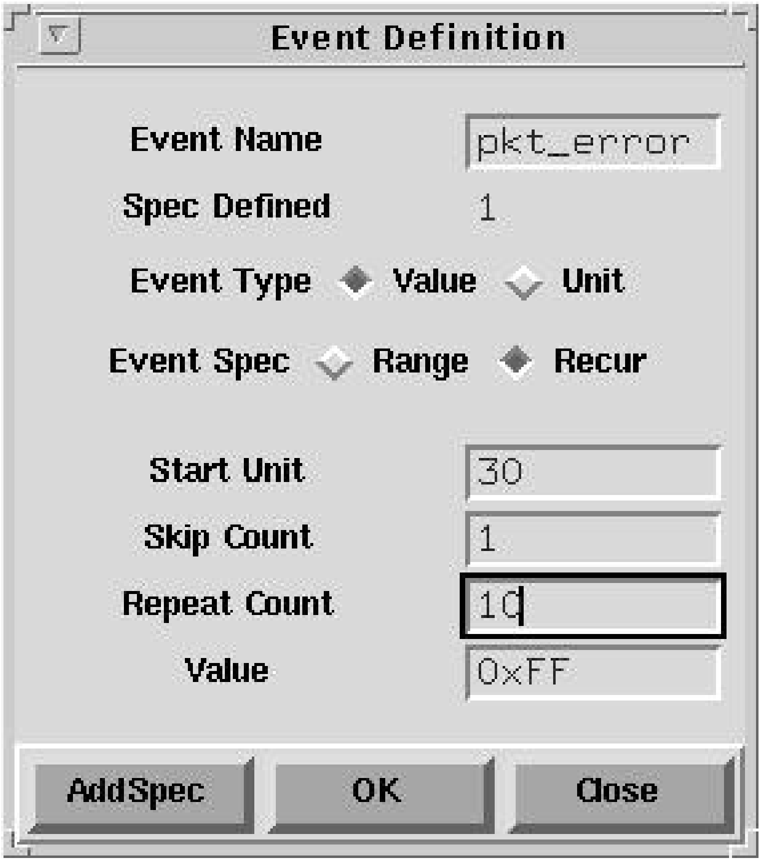

Figure 3-16b. Event Definition Menu Panels (Cont'd)

OK and subsequently CLOSE, will save the input and return the user to the Menu Panel shown in Figure 3-7. Depressing OK once more will return the user to the CADU Stream graphical display shown in Figure 3-3.

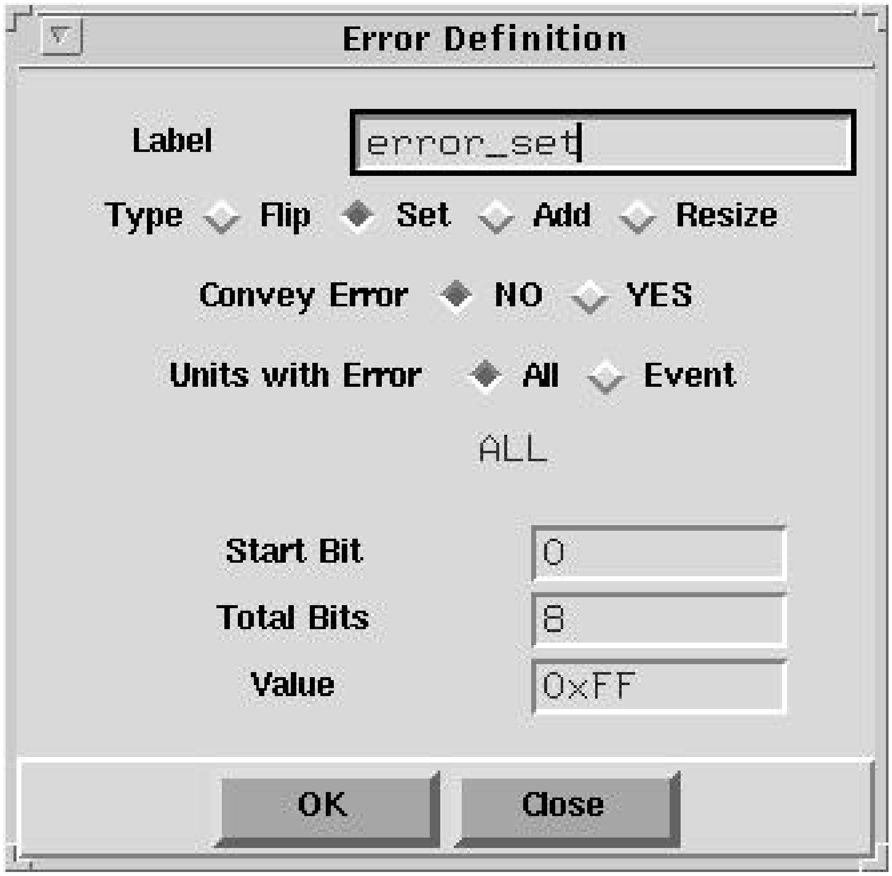

Figure 3-17. Set Error Definition Menu Panel

The next step in the process is to define the Packet multiplexing strategy that will define the stream packet source for the frames designated by the selected VCID stream.

Clicking on any one of the 'vcxmux' button, in the menu panel shown in Figure 3-3, will display the Packet Multiplexer Menu Panel for the selected VCID, shown in Figure 3-18. Similar to the Frame Multiplexing strategy, the Packet Multiplexing strategy can either use the 'Set Range' option for interleaving the order of packets within the 'vc' stream, or the 'SetPattern' option to select a recurring pattern. As in the case of the Frame Multiplexing menu, there is an option to read in a file that lays out the complete interleaving pattern within the virtual channel, by invoking the MUXtool option. The GUI does not support this option, but, the user can manually insert the arguments to use the MUX tool option as described in the script language description in the latter part of this document. Once the multiplexing strategy is entered, the View button, elongates the Menu Panel, showing the user the composition of the 'vc1mux' stream by APID and the percentage of each 'apxxx' within the 'vc1mux' stream. The CLOSE option takes the user back to the graphical representation of the CADU stream, as shown in Figure 3-3.
The next step in the process is to define the packets within each APID stream. This is done by clicking on a selected packet stream, 'apxxxx', displays the Packet Definition Menu panel shown

in Figure 3-19. The first three entries in this panel are already entered. The Version number is entered, which for EOS-AM1 is Version 1. The Max entry is used to define the max number of packets that could be created from this APID stream. This is especially important when reading from a raw graphics file, in which the size is fixed. The option to create Telecommand packets is available. For return link data, this option is set to 'No'. The Packet Data has to be defined, and by selecting the 'Yes' option for Packet Data, the Data Definition Menu Panel as shown in Figure 3-20a is displayed.

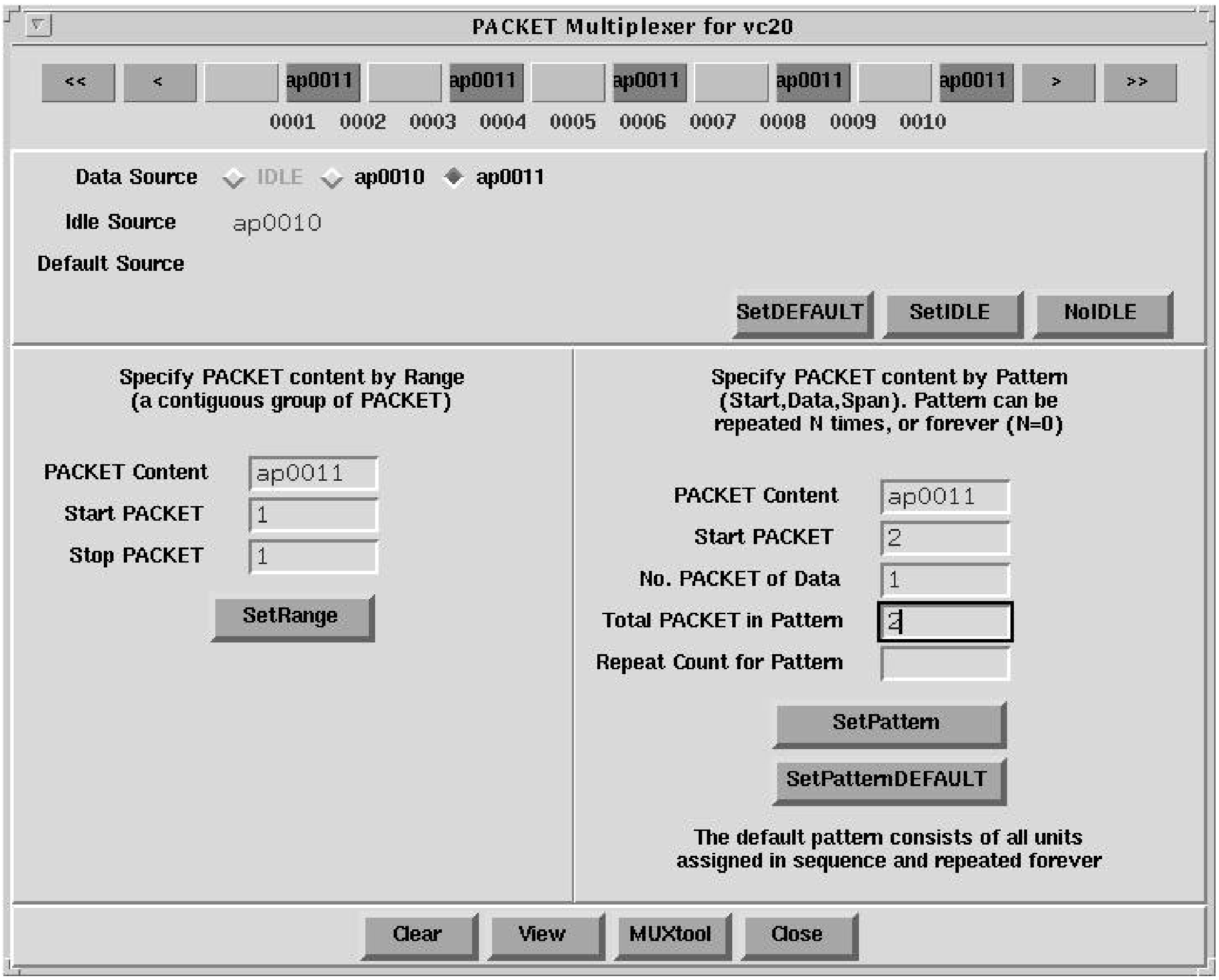

Figure 3-18. Packet Multiplexer Menu Panel

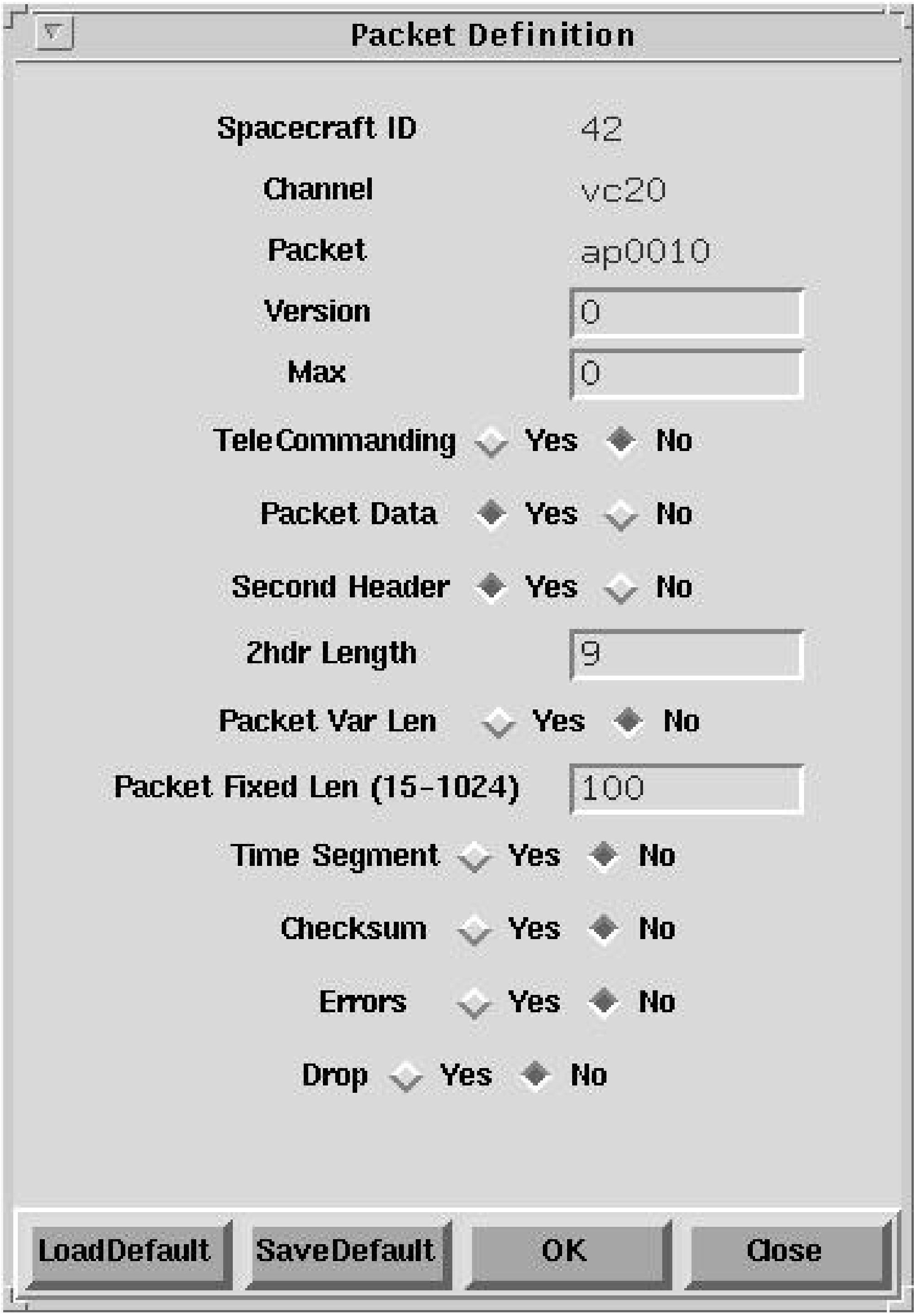

Figure 3-19. Packet Definition Menu Panel

If the Basic option is selected, only the default header definitions for time code can be entered for the packet stream. If the optional button is selected, in addition to the default value for the time code, which in this application for SCTGEN is tcEDOS, three other options are selectable. The

contents of the data region can be either of a 'Fixed' pattern; 'Step' pattern; 'Raw File' or a 'Random' pattern. When the 'Raw File' option is selected, the menu panel shown in Figure 3-20b is displayed, where the name and path of the file have to be entered. In the 'Fixed' and 'Step' options the values have to be entered. OK and subsequently CLOSE, will save the input and return the user to Figure 3-19.

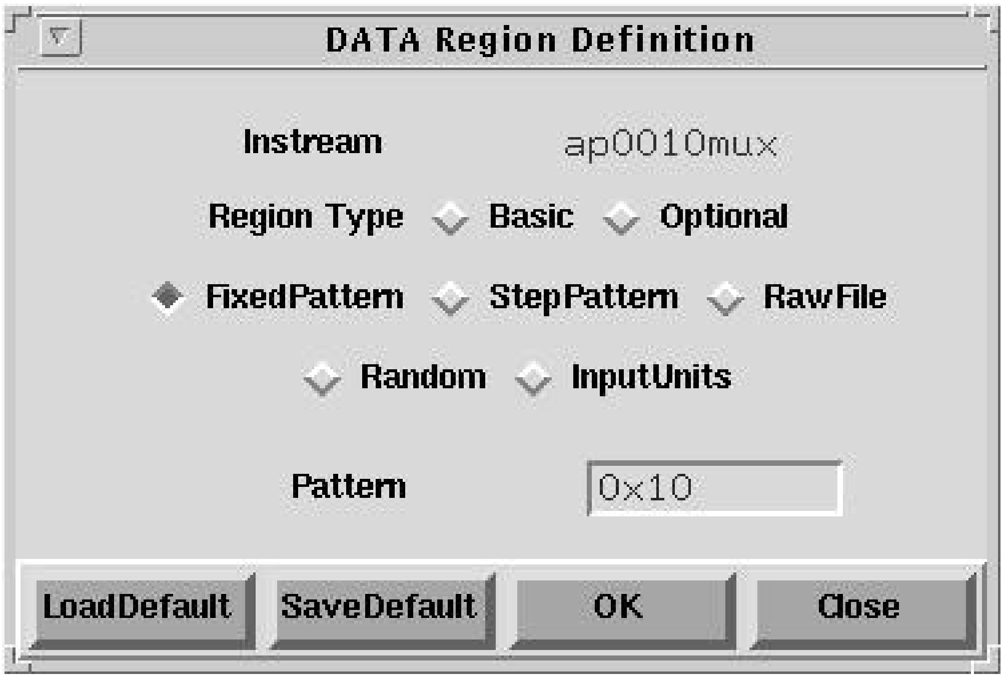

Figure 3-20a. Data Definition Menu Panel

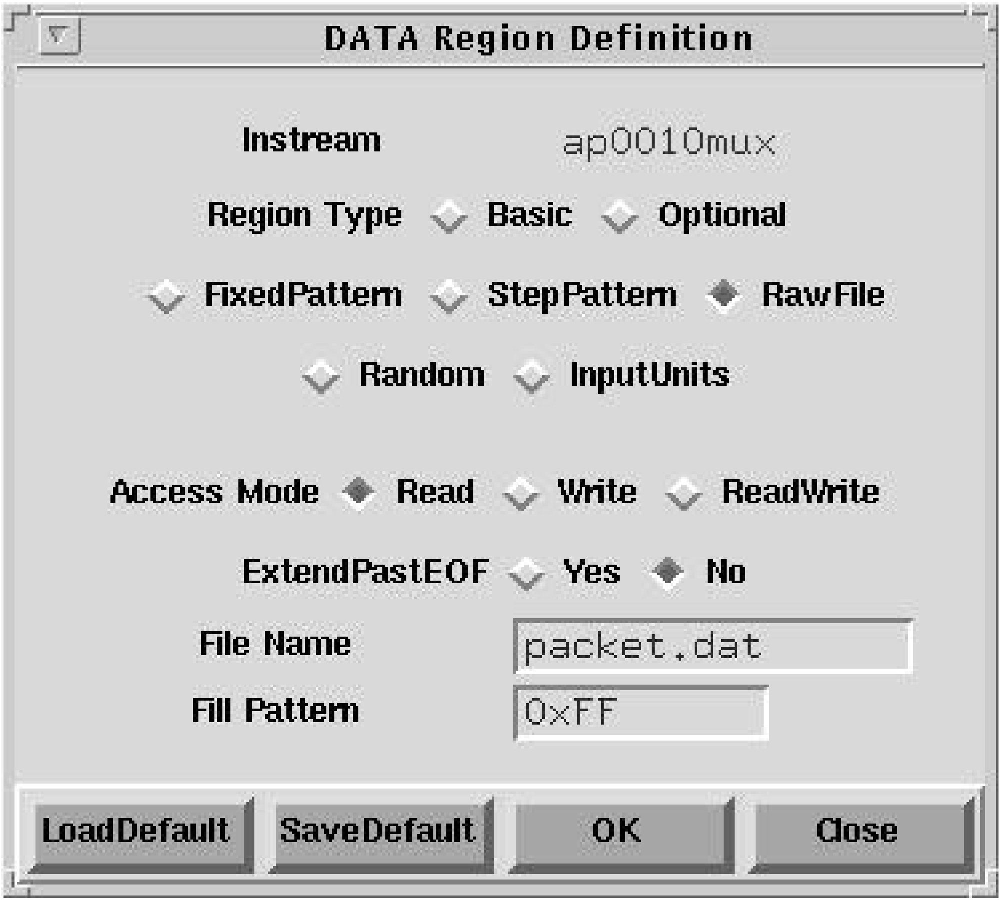

Figure 3-20b. Data Definition Menu Panel (Cont'd)

To define the time-code within the packet header, the Second Header option is selected. This displays the 2HDR Definition Menu panel, shown in Figure 3-21. For this application, SCTGEN has been customized for EOS-AM, and thus the tcEDOS option is used to initialize and define the step size for the time-code. The 'Day' entry defines the start day (Julian calendar) for the packet stream. The 'MS of Day' initializes the start time in milliseconds of the day when the packet stream should start, and 'MICRO of MS' initializes the micro-second granularity in the start time. The next two entries define the step in milliseconds and microseconds between consecutive packets within the APID or packet stream. The user is provided with the capability of defining a 'Ramp' function for the time code if required, and the correction due to Drift. The values that can be entered in this boxes are described in the script description in later sections of this document.

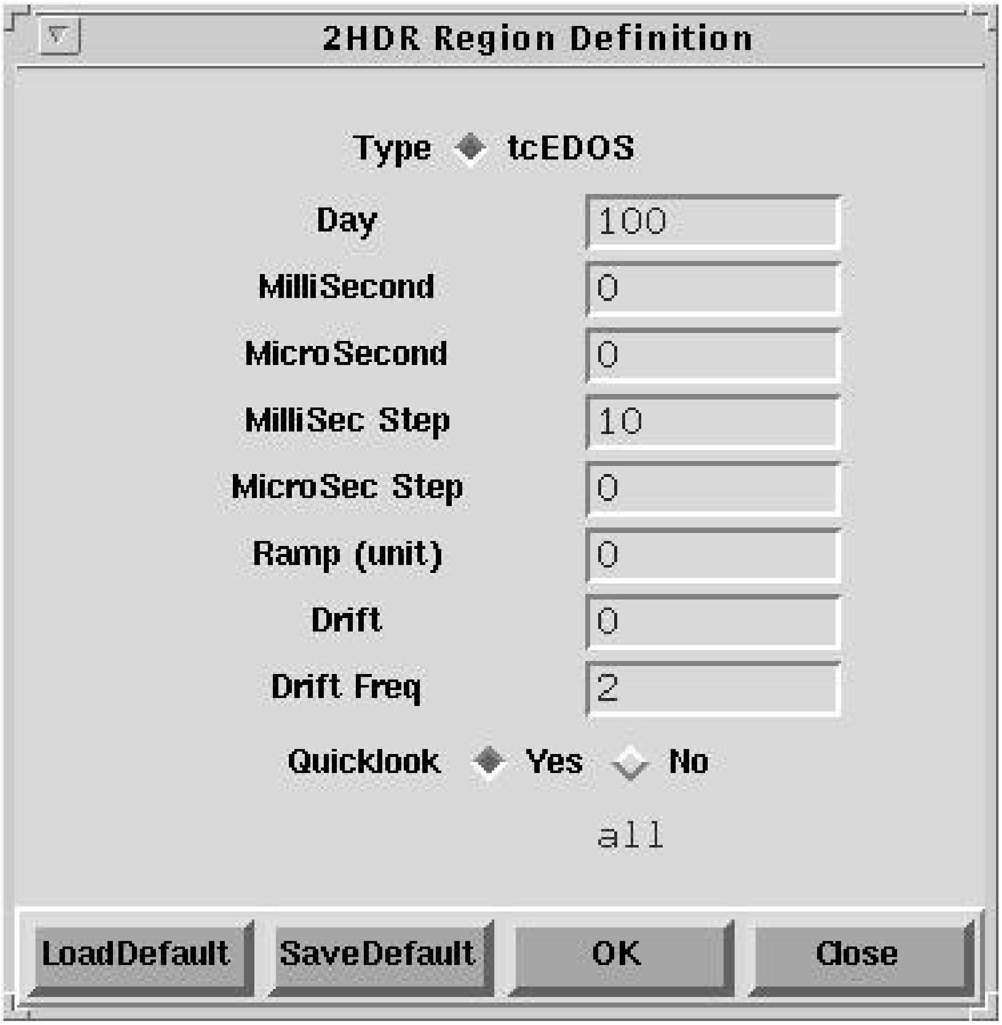

Figure 3-21. Secondary Header Definition Menu Panel

OK and subsequently CLOSE, will save the input and return the user to Figure 3-19. The next entry on the packet definition menu panel is the option of having variable length packets. This option will be used for Telecommand packets and will be described in the later sections. When the 'No' option is selected for 'Packet Variable Length', a value for the 'Packet Fixed Length' has to be entered by the user. The 'Checksum' option is either selected or not, and when selected this performs a checksum on each packet and inserts the checksum value as the last byte in the packet data region. The Error' insertion process follows the same steps used for the error insertion at the frame level, and the steps are repeated here for completeness.

Selecting the 'Yes' option on the next entry in the Packet Definition, will display the Error Listing Menu Panel, shown in Figure 3-22.

For all APIDs, any number of errors maybe introduced. In the Error Listing Menu, the AddError is depressed to display the default Error Definition menu panel, shown in Figure 3-23, which comes up for the 'Flip' error type. In this menu, the user can label the type of error; select the method by which the error is to be inserted, i.e., by flipping bits, or setting bits to specific values, or by completely dropping packets identified by the APID index or count.

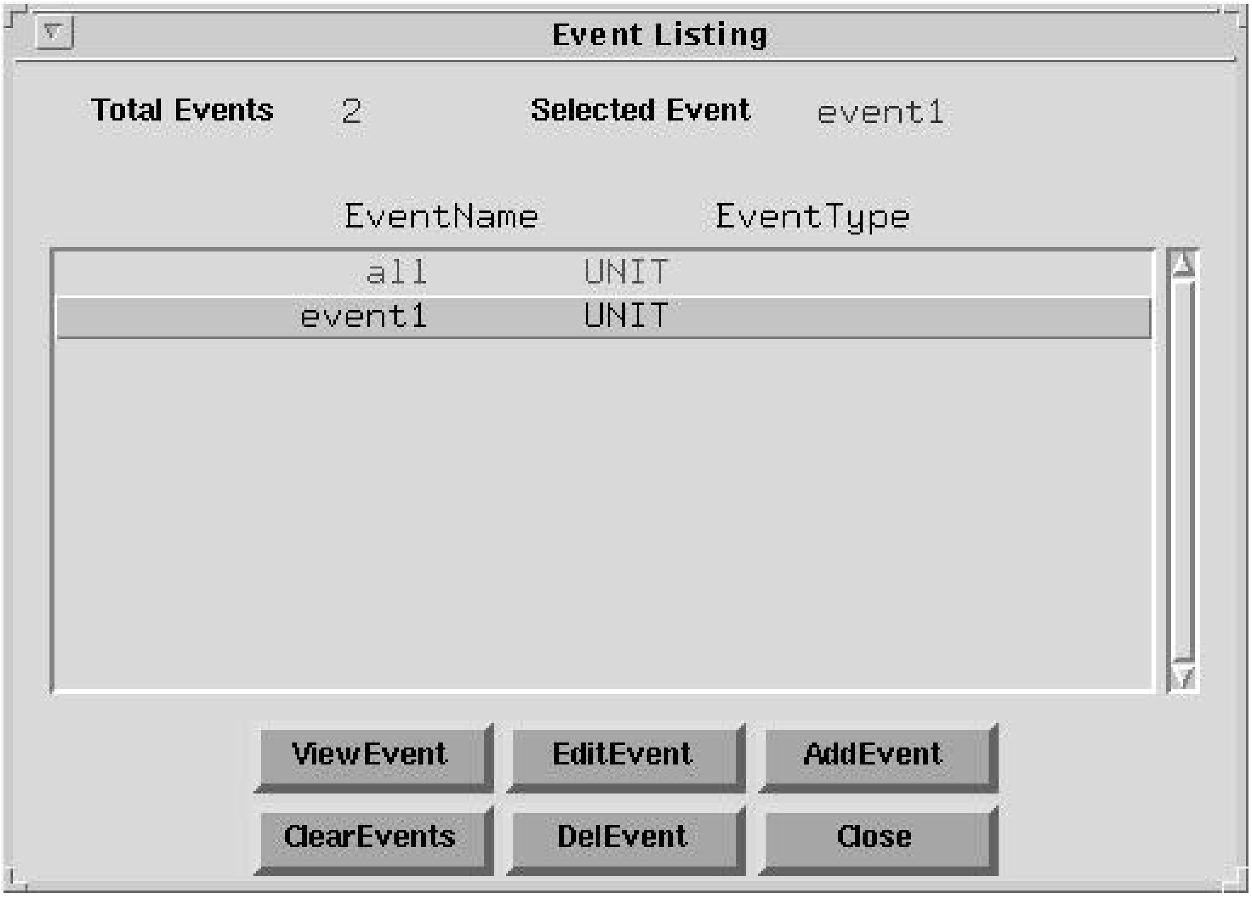

Figure 3-22. Error Listing Menu Panel

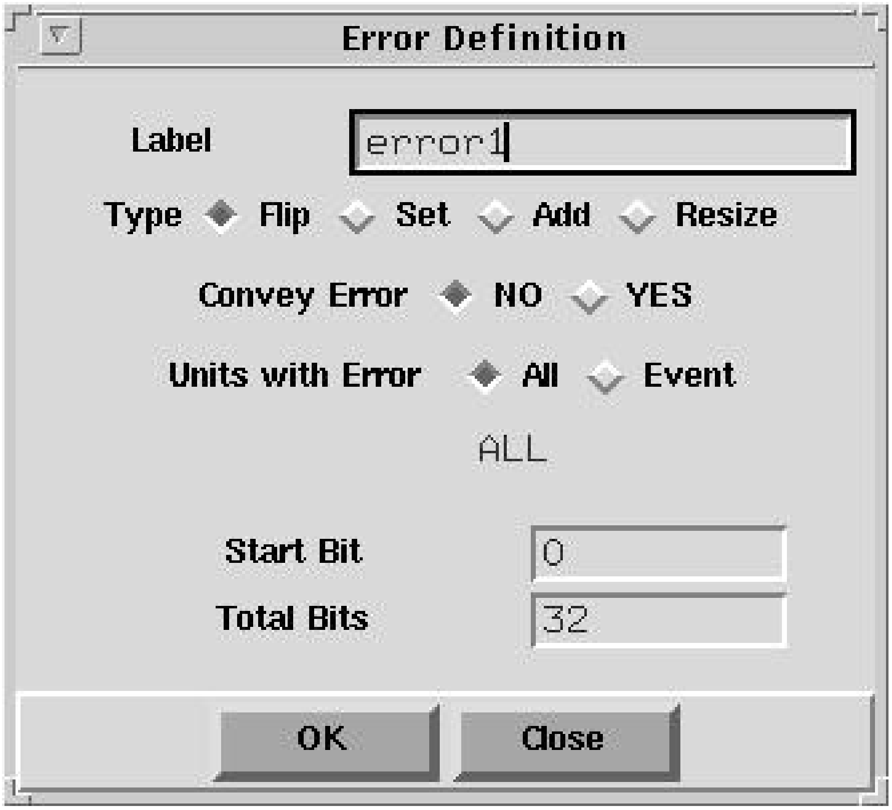

Figure 3-23. Flip Error Definition Menu Panel

When bits are to be flipped, Figure 3-23, the exact location of the bits within the packet, and the number of bits to be flipped are specified. The user can also specify whether this error event is to be implemented on all the packets from this APID stream or certain packets or ranges of packets as specified by their index in the stream. If all the packets are selected to have the error then the 'All' option is selected. If specific units are ear-marked for error insertion, then the 'Event' option is selected which displays the Event Listing Menu Panel, shown in Figure 3-24.

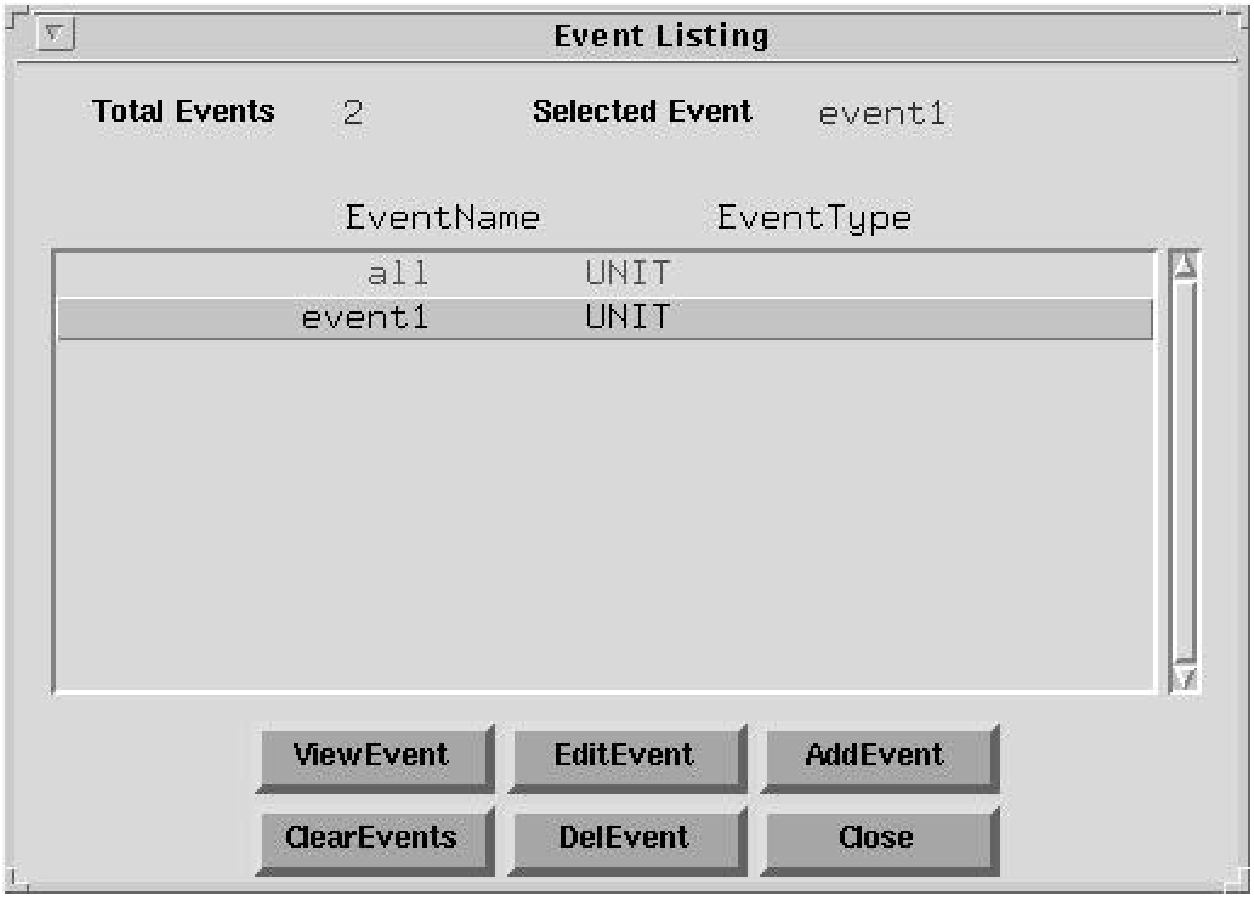

Figure 3-24. Event Listing Menu Panel

The AddEvent option displays the Event Definition Menu Panels, shown in Figure 3-26, that allow the user to select the packets for error insertion. The Event name is user selectable and the occurrence of the event can be either a value or a unit. The Event Spec defines whether a certain range of units or packets specified by their unit number in the stream, Figure 3-25a; or a recurrent pattern is specified based on the start packet number; how many contiguous packets from this stream; how many packets are to be skipped; and how many times this pattern is to be repeated, Figure 3-25b. The section on the script descriptions will provide examples of both options. Once the event is defined, the 'Add Spec' button MUST be depressed for the event to be registered, and then the OK and CLOSE buttons in that order will take the user back to the Event Listing Menu in Figure 3-24. The CLOSE options will save the selected input and return the user to the Error Definition Menu Panel.

Similarly when bits are set to certain values, the user can select the label, occurrence, location and quantity of the bits and the value to be set, Figure 3-26. In dropping Packets, the user can select packets by their index or unit number in the VCID stream. The Convey function is optionally set to convey the error up to the RS encoding process. If this is not set, the RS encoding process will correct the errors inserted in the Packet, and the created test data will not contain the Packet level errors.

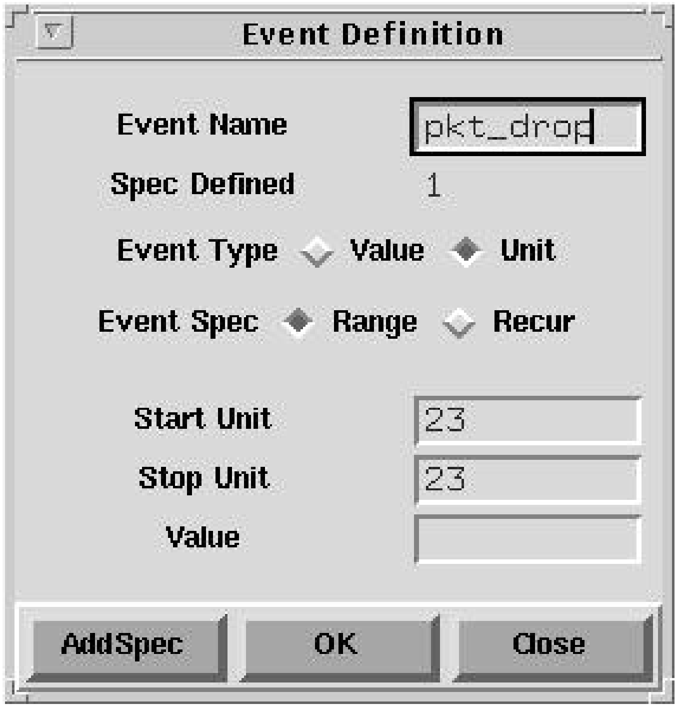

Figure 3-25a. Event Definition Menu Panels

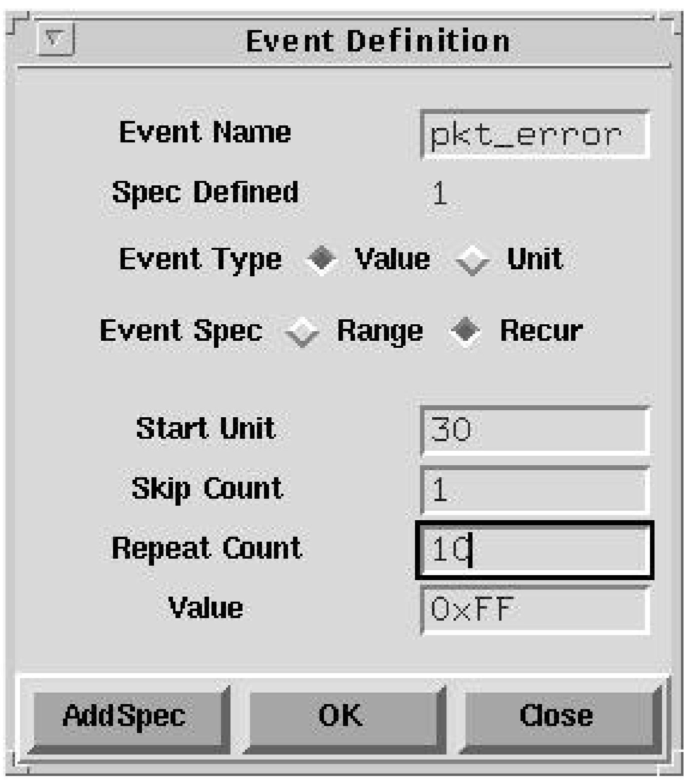

Figure 3-25b. Event Definition Menu Panels (Cont'd)

The other buttons shown in Figure 3-24 are self-explanatory. The DelError button allows the user to selectively delete an error entry, and finally the Clear Error button clears out all the listed errors.
OK and subsequently CLOSE, will save the input and return the user to the Menu Panel shown in Figure 3-19. Depressing OK once more will return the user to the CADU Stream graphical display shown in Figure 3-3.

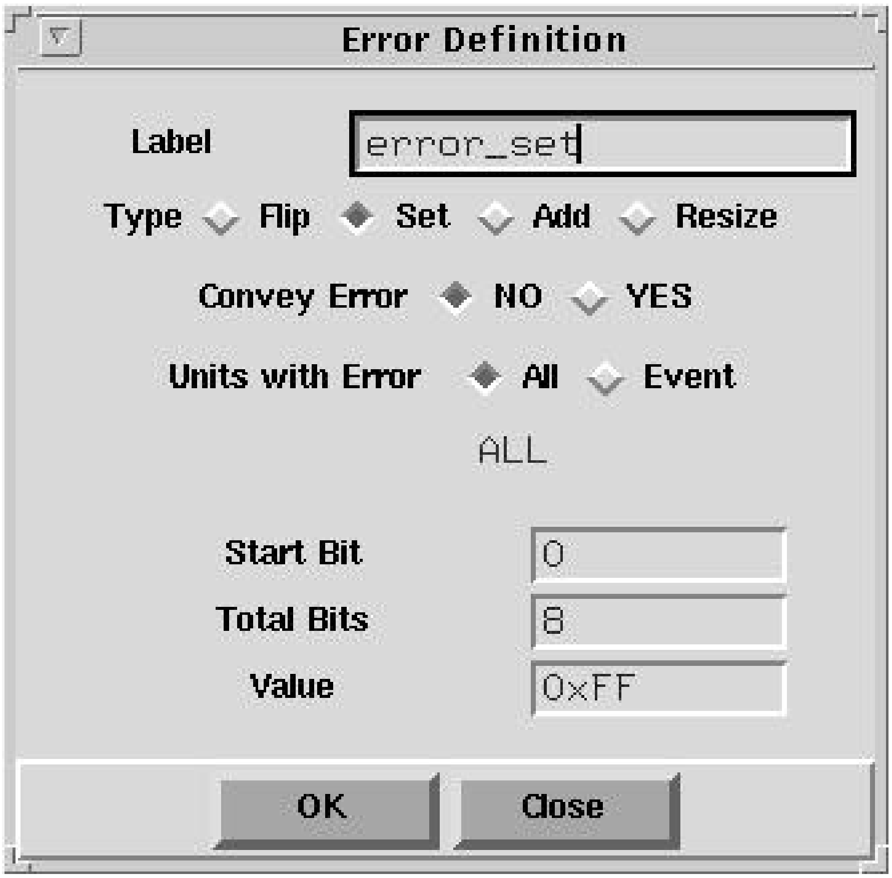

Figure 3-27. Set Error Definition Menu Panel

At each stage in the hierarchy, when the user has completed entering the values needed for data generation and saved (or 'OK'ed the same), the box changes color from blue to green. When the box is selected it changes color from blue to yellow. Once the whole data scenario has been entered, the user can select the 'Script' option, whereby the Script Window is displayed, as shown in Figure 3-27a.

The script created has been commented to illustrate where the different argument values were input by the user and how they should appear in the script. The commented script is shown as Figure 3-27b.

In this version of the SCTGEN GUI, the user can not use the error listing menu panel to view or edit previously specified errors. One option is to view the errors in the generated scripts using the script window and manually editing the error specification. The other option is to delete the selected error and redefine the error.

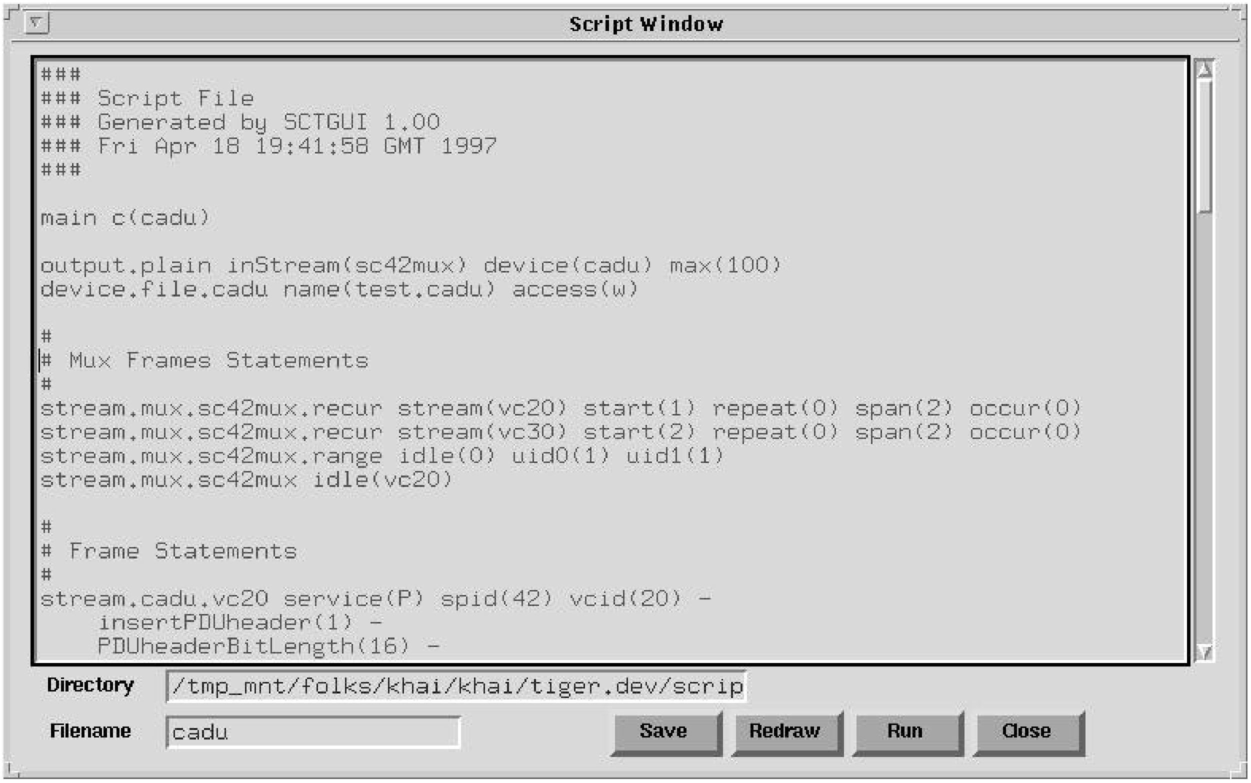

Figure 3-27a. Script Window

```
###
### Script File
### Generated by SCTGEN GUI 0.15
###
### Thu Dec 12 18:34:27 GMT 1996
###

* the first line main c(cadu) is created when the cadu scenario is selected *
       1. main c(cadu)

* the output definition menu creates the second line *
       2. output.plain inStream(sc42mux) device(cadu) max(100)

* the device definition menu creates the third line *
       3. device.file.cadu name(cmtest3err.cadu) access(w)


# MuxFRAMErangeScript

# MuxFRAMEpatternScript

* the frame multiplexer menu for sc42 creates the fourth and fifth lines *
       4. stream.mux.sc42mux.recur stream(vc1) start(1) repeat(0) span(2) occur(0)
       5. stream.mux.sc42mux.recur stream(vc2) start(2) repeat(0) span(2) occur(0)

# FrameScript

* the frame definition menu for vc1 and within this menu, the service definition, sync
definition, and RS definition menus create the sixth line *
       6. stream.cadu.vc1 service(P) spid(42) vcid(1)  -
              insertPDUheader(1) PDUheaderBitLength(16)  -
              sync(1) frameSync(0x1ACFFC1D) syncBytes(4)  -
              RSencode(1) RSinterleave(4) RSdual(1) RSempty(0)  -
              length(1024) max(1000)

* the frame definition menu for vc2 and within this menu, the service definition, sync
definition, and RS definition menus create the seventh line *
       7. stream.cadu.vc2 service(P) spid(42) vcid(2)  -
              insertPDUheader(1) PDUheaderBitLength(16)  -
              sync(1) frameSync(0x1ACFFC1D) syncBytes(4)  -
              RSencode(1) RSinterleave(4) RSdual(1) RSempty(0)  -
              length(1024) max(1000)

# RegionScriptFRAME

* the frame definition menu for vc1 and within this menu, the service definition menu,
creates the eighth line *
       8. stream.cadu.vc1.region.data type(C) inStream(vc1mux) lastUnit(I) -
              ERcomposition(1)

* the frame definition menu for vc1 and within this menu, the service definition menu,
creates the ninth line *
       9. stream.cadu.vc2.region.data type(C) inStream(vc2mux) lastUnit(I) -
              ERcomposition(1)

# MuxPACKETrangeScript
# MuxPACKETpatternScript

* the packet multiplexer menu for vc1 creates the tenth and eleventh lines *
       10. stream.mux.vc1mux.recur stream(ap0012) start(1) repeat(0) span(2) occur(0)
       11. stream.mux.vc1mux.recur stream(ap0013) start(2) repeat(0) span(2) occur(0)

* the packet multiplexer menu for vc2 creates the twelfth and thirteenth lines *
       12. stream.mux.vc2mux.recur stream(ap0021) start(1) repeat(0) span(2) occur(0)
       13. stream.mux.vc2mux.recur stream(ap0022) start(2) repeat(0) span(2) occur(0)

# PacketScript

* the packet definition menu for ap0012 creates the fourteenth line *
       14. stream.pkt.ap0012 appid(0012)  -
              version(1) max(100000) length(100)  -
              2hdr(1) 2hdrLength(9)
```

Figure 3-27b Commented Script (Cont'd)

```
* the packet definition menu for ap0013 creates the fifteenth line *

       15. stream.pkt.ap0013 appid(0013)  -
              version(1) max(100000) length(200)  -

              2hdr(1) 2hdrLength(9)

* the packet definition menu for ap0021 creates the sixteenth line *
       16. stream.pkt.ap0021 appid(0021)  -
              version(1) max(100000) length(300)  -
              2hdr(1) 2hdrLength(9)

* the packet definition menu for ap0022 creates the seventeenth line *
       17. stream.pkt.ap0022 appid(0022)  -
              version(1) max(100000) length(400)  -
              2hdr(1) 2hdrLength(9)

# RegionScriptPACKET

* the packet definition menu for ap0012 and within this menu, the packet data definition, and
secondary header definition menus create the eighteenth and ninteenth lines *
       18. stream.pkt.ap0012.region.2hdr type(tcEDOS)  -
              day(0) msOfDay(0) microOfMs(0) msStep(10) microStep(0)  -
              ramp(0) drift() driftFreq()
       19. stream.pkt.ap0012.region.data type(F)  -
              pattern(0x01)

* the packet definition menu for ap0013 and within this menu, the packet data definition, and
secondary header definition menus create the twentieth and twenty-first lines *
       20. stream.pkt.ap0013.region.2hdr type(tcEDOS)  -
              day(0) msOfDay(0) microOfMs(0) msStep(10) microStep(0)  -
              ramp(0) drift() driftFreq()
       21. stream.pkt.ap0013.region.data type(F)  -
              pattern(0x11)

* the packet definition menu for ap0021 and within this menu, the packet data definition, and
secondary header definition menus create the twenty-second and twenty-third lines *
       22. stream.pkt.ap0021.region.2hdr type(tcEDOS)  -
              day(0) msOfDay(0) microOfMs(0) msStep(10) microStep(0)  -
              ramp(0) drift() driftFreq()
       23. stream.pkt.ap0021.region.data type(F)  -
              pattern(0x20)

* the packet definition menu for ap0022 and within this menu, the packet data definition, and
secondary header definition menus create the twenty-fourth and twenty-fifth lines*
       24. stream.pkt.ap0022.region.2hdr type(tcEDOS)  -
              day(0) msOfDay(0) microOfMs(0) msStep(10) microStep(0)  -
              ramp(0) drift() driftFreq()
       25. stream.pkt.ap0022.region.data type(F)  -
              pattern(0x21)
#
# Error statements
#
* the frame definition menu for vc1 and within this menu, the error listing definition, and
set error definition menus create the twenty-sixth line *
       26. stream.cadu.vc1.error.set label(frmsyn) convey(0) event(1) -
              startbit(8) bits(8) v(5)

* the frame definition menu for vc2 and within this menu, the error listing definition, and
flip error definition menus create the twenty-seventh line *
       27. stream.cadu.vc2.error.flip label(frmver) convey(0) event(1) -
              startbit(33) bits(8)

* the packet definition menu for ap0012 and within this menu, the error listing definition,
and flip error definition menus create the twenty-eighth line *
       28. stream.pkt.ap0012.error.flip label(pkttime) convey(0) event(1) -
              startbit(49) bits(8)

* the packet definition menu for ap0021 and within this menu the error listing definition,
and set error definition menus create the twenty-ninth line *
       29. stream.pkt.ap0021.error.set label(pktver) convey(0) event(1) -
              startbit(1) bits(16) v(9)
#
# Event statements
#
* no other event statements were created *
```

Figure 3-27c Commented Script (Cont'd)

3.3 FORWARD LINK DATA SCENARIO DEVELOPMENT

Similar to the Return Link Data development, the first step is to develop a scenario for data generation. Choosing the 'new' option on the scenario pull-down menu, will open a window that accepts the user input. The following paragraphs will describe the development of a typical Forward Link Scenario. In the menu depicted in Figure 3-28, the Telecommand or TC option is selected and subsequently the scenario ->new option is selected. The TC Data Scenario Menu Panel is displayed, as shown in Figure 3-29.

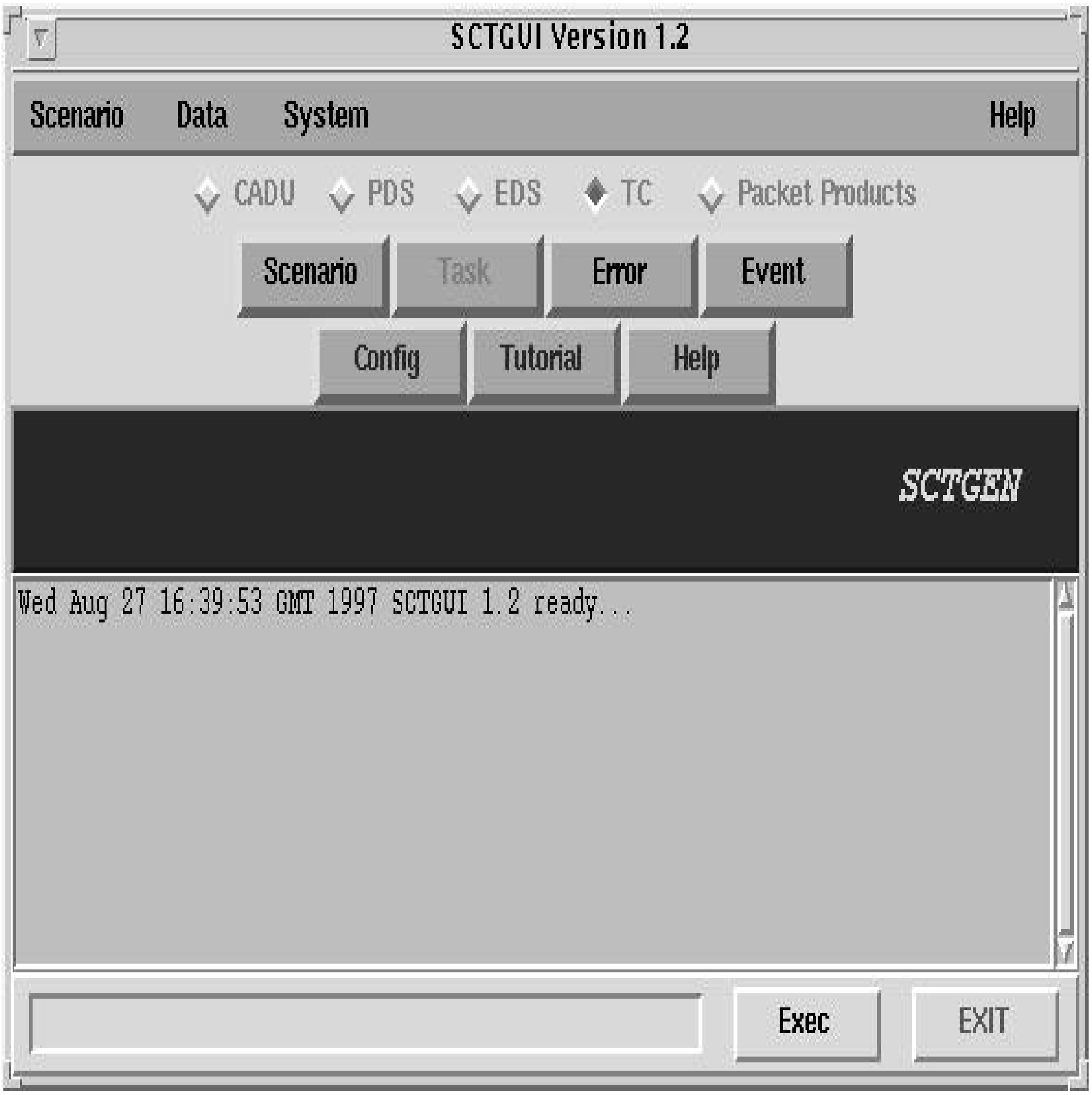

Figure 3-28. SCTGEN Main Menu Panel

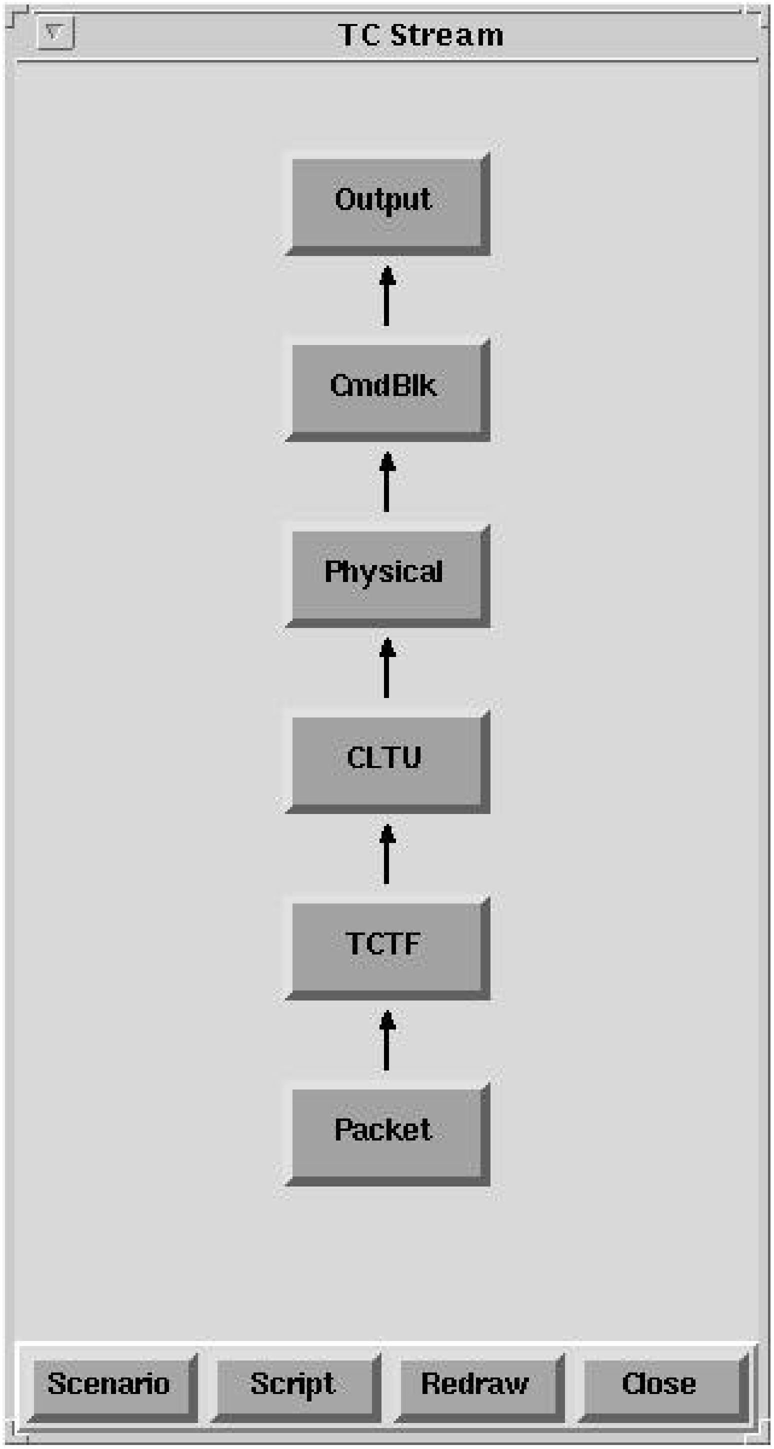

Figure 3-29. Telecommand Data Scenario Menu Panel

From this point on each of the icons shown on the TC stream menu open up to more detailed windows, for example: 'Output', will open the TC Output Definition Menu Panel shown in Figure 3-30, which will allow the user to define the number of units, which in this case is TC blocks, the format of the File being created, and the Device that is to be generated as a product. For Output type, the user should select File. The other three options are not applicable with this version of the software. When 'File' is selected, this means that the output that is created is a plain file. The instream name in this scenario is the name of the block immediately below the 'Output' block in the TC Data Scenario menu panel, namely the 'CmdBlk' block. The number of units in this case defines the number of Telecommand Transfer frames the user wants in this test data file. The File format defines whether the output file has a header and trailer or is just a plain stream of frames. Again, in the case of ETS the forward link data is in a plain format. The File size is applicable when the test data to be created as a number of files, the size of each file can be selected.

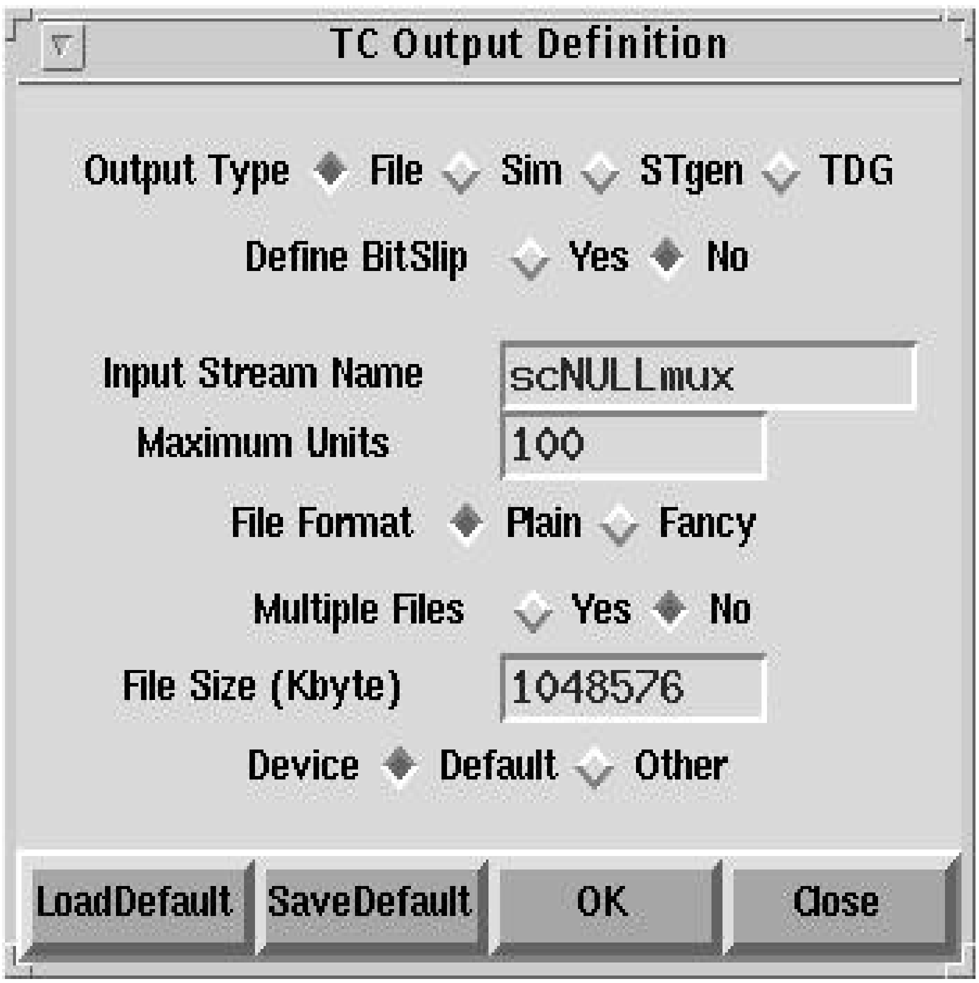

Figure 3-30. TC Output Definition Menu Panel

To define a file name Device should be selected by depressing the 'other' button. This will bring up the next menu panel, Figure 3-31 Device Definition. Here the Access mode is defined to be 'write', and the file name and extension are user selectable. The Device type implements either the creation of the test data file or the pseudo-generation of the data file, i.e.,, if the user wants to see the effects of the ordering and error insertion strategy without actually generating the data, the 'null' option allows only the Expected Results file to be created, with an extension of 'filename.er'. Note, the option to create an Expected Results File has to be specified.

Once the Device is defined, the OK button must be depressed to save the input at this level. This will take the user back to the Output Definition Menu, where once again the OK button has to depressed to save the input at that level. Depressing the close button will nullify the input entered by the user.

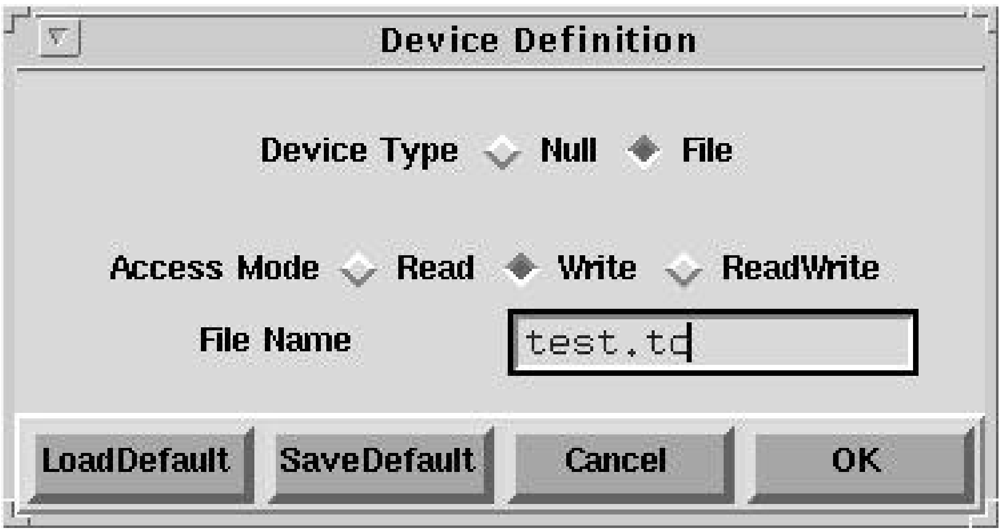

Figure 3-31. Device Definition Menu Panel

Closing this menu will take the user back to the graphical menu where the next button, 'CmdBlk' in the hierarchchy is depressed. This button will open up the CmdBlk Definition Menu Panel, as shown in Figure 3-32.

Figure 3-32 defines the CmdBlk Stream for the output TC Stream. This definition for the EDOS Ground Message Header, will not require the user to calculate the length of the units that are being fed by the instream 'Physical' block. In this version three different regions of the CmdBlk are defined. The data region specifies the instream, which in this case is defined as the 'Physical' block; the fixed values that are user selectable; and finally the PB5 time stamp region which allows day and time calculated from a user-selectable epoch and the respective steps in these time units to be initialized.

The 'MessageType' , 'Source', Destination' is user-selected with reference to the EDOS-EGS Interface Control Document. The SPID for EOS-AM1 is '42', i.e., 2-byte field, and the 'StartSequence' is defaulted to start at '0'. The 'Major' and 'Minor' versions of the software are user-selectable, and for ETS are defaulted to '0' and '0'. The 'Max Units' field specifies the number of command blocks to be created and is user-selectable. The stream is validated as being the 'CmdBlk' and the instream is specified according to the scenario selected where the next block in the hierarchy is the 'Physical' stream, as validated in the next region to be defined.

Once the Ground Message Header is defined the user selects the data region of the CmdBlk Menu Panel, which open up the Data Region Definition, shown in Figure 3-33. The data region of the CmdBlk definition allows the user to specify the instream, which in this case validates the 'Physical' as the next block in the hierarchy, and the integral number of units to encapsulate.

OK and CLOSE returns the user back to the menu panel shown on Figure 3-32. The next step in the CmdBlk definition is the definition of the PB5 Time Stamp. The PB5 is defaulted to with some predefined values. If the user wants to change the PB5 values, the 'Yes' button has to be selected. When the 'Yes' option is selected, the PB5 Region Definition Menu Panel, Figure 3-34 is displayed. The PB5 Region Definition describes these options and the sample script at the end of this section shows the script lines that are produced when the CmdBlk is defined using this menu panel. For this application, SCTGEN has been customized for EOS-AM, and thus the 'pb5' option is used to initialize and define the step size for the time-stamp that appears in the EDOS Ground Message Header.

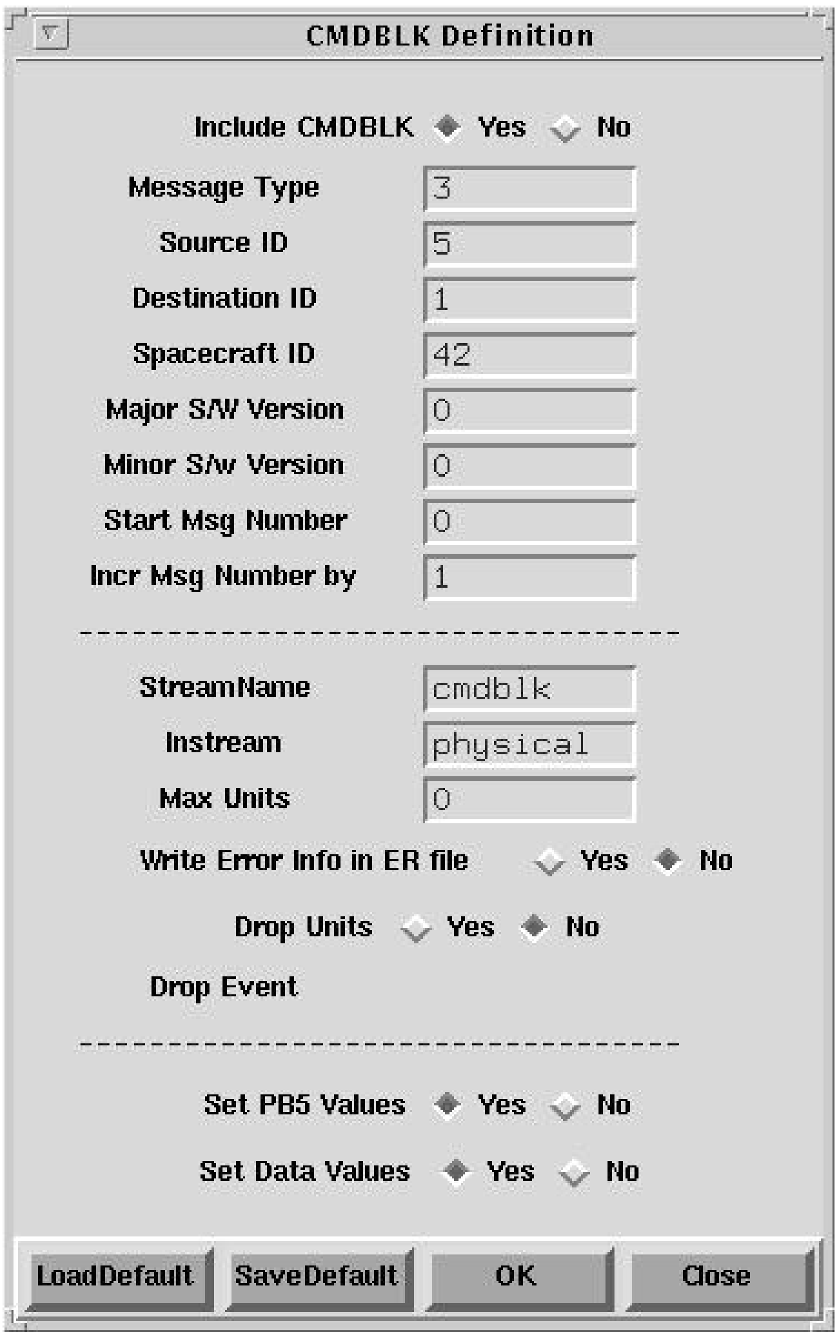

Figure 3-32. Command Block Definition Menu Panel

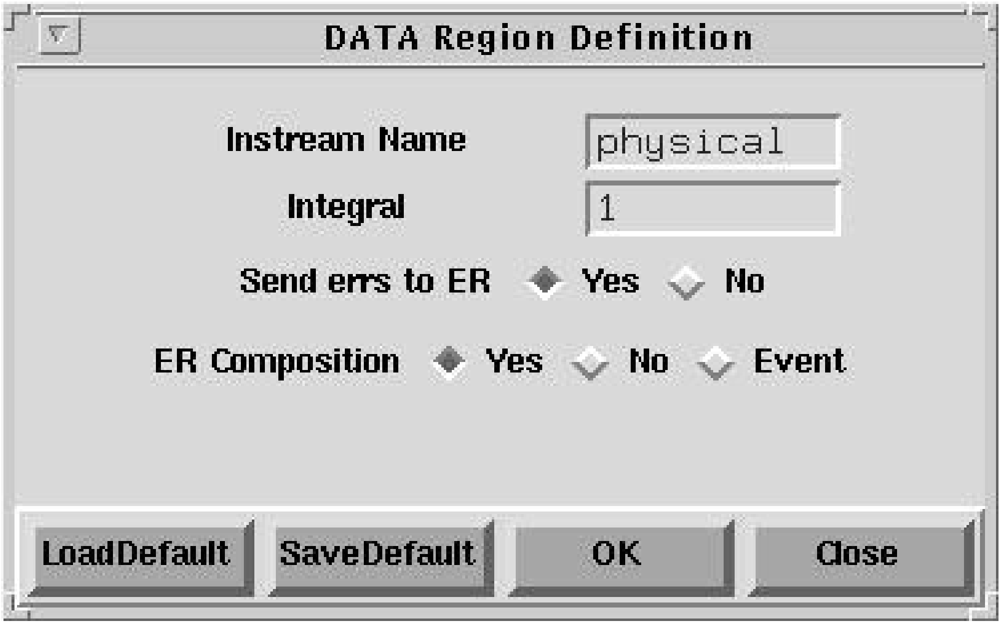

Figure 3-33. Command Block Definition Menu Panel

The 'Day' entry defines the start day (Julian calendar) from a user selected epoch for the TC stream. The default that has been calculated and set is from October 10, 1995. The 'DayStep' defines the day increment and the default is 1 day at a time. The 'Seconds' initializes the start time in seconds of the selected day, the 'Milliseconds' initializes the milliseconds and the 'Microseconds' initializes the microseconds of the selected day of the TC stream time stamp. The next three entries, 'SecondsStep', 'MilliStep', and 'MicroStep' define the step in seconds, milliseconds and microseconds between consecutive TC blocks within the TC stream.

The user is provided with the capability of defining a 'Ramp' function for the time stamp if required, and the correction due to Drift. When the 'Ramp' option is selected, the user is prompted to specify the number of units, in this case Command Blocks, that will have the same time stamp, before the time is stepped up to the next defined value. The 'Drift' and 'Drift Frequency' allows the user to introduce a non-uniform step in time at selected units. OK and CLOSE returns the user back to the CmndBlk Menu in Figure 3-32.

The option to write the Error information into the ER file is selected at this stage by clicking on the 'Yes' button. Selected units may be tagged as dropped, such that the resulting instream will be devoid of specific units as identified by their index or sequence count. The Drop Event is defined by selecting the 'Drop Units' option, whereby the Event Listing Menu Panel, shown in Figure 3-35, is displayed. The Event Definition and the associated menus are described in the following paragraphs. Depressing the OK button saves the input and CLOSE returns the user to Figure 3-29.

The AddEvent option displays the Event Definition Menu Panels, shown in Figure 3-36a, that allow the user to select the CmdBlks to be dropped. The Event name is user selectable and the occurrence of the event can be either a value or a unit.

The Event Spec defines whether a certain range of units specified by their unit number in the stream, Figure 3-36a; or a recurrent pattern is specified based on the start unit number; how many contiguous units from this stream; how many units are to be skipped; and how many times this pattern is to be repeated, Figure 3-36b. The section on the script descriptions will provide examples of both options. The OK and CLOSE options will save the selected input and return the user to the CmdBlk Definition Menu Panel. OK and subsequently CLOSE, will save the input and return the user to the CmdBlk Definition Menu Panel shown in Figure 3-32.

Figure 3-34. PB5 Region Definition Menu Panel

The OK and CLOSE options will save the selected input and return the user to the Main TC Stream Menu Panel. Depressing 'Physical' allows the user to move to the next block in the heirarchy to be defined. This displays the Physical Definition Menu Panel shown in Figure 3-37. This menu will enable the user to define the Acquisition bits in the Physical steam that will be pre-pended to the instream, which in this case is 16 bytes, and 8 idle bits appended, which is shown in Figure 3-37.

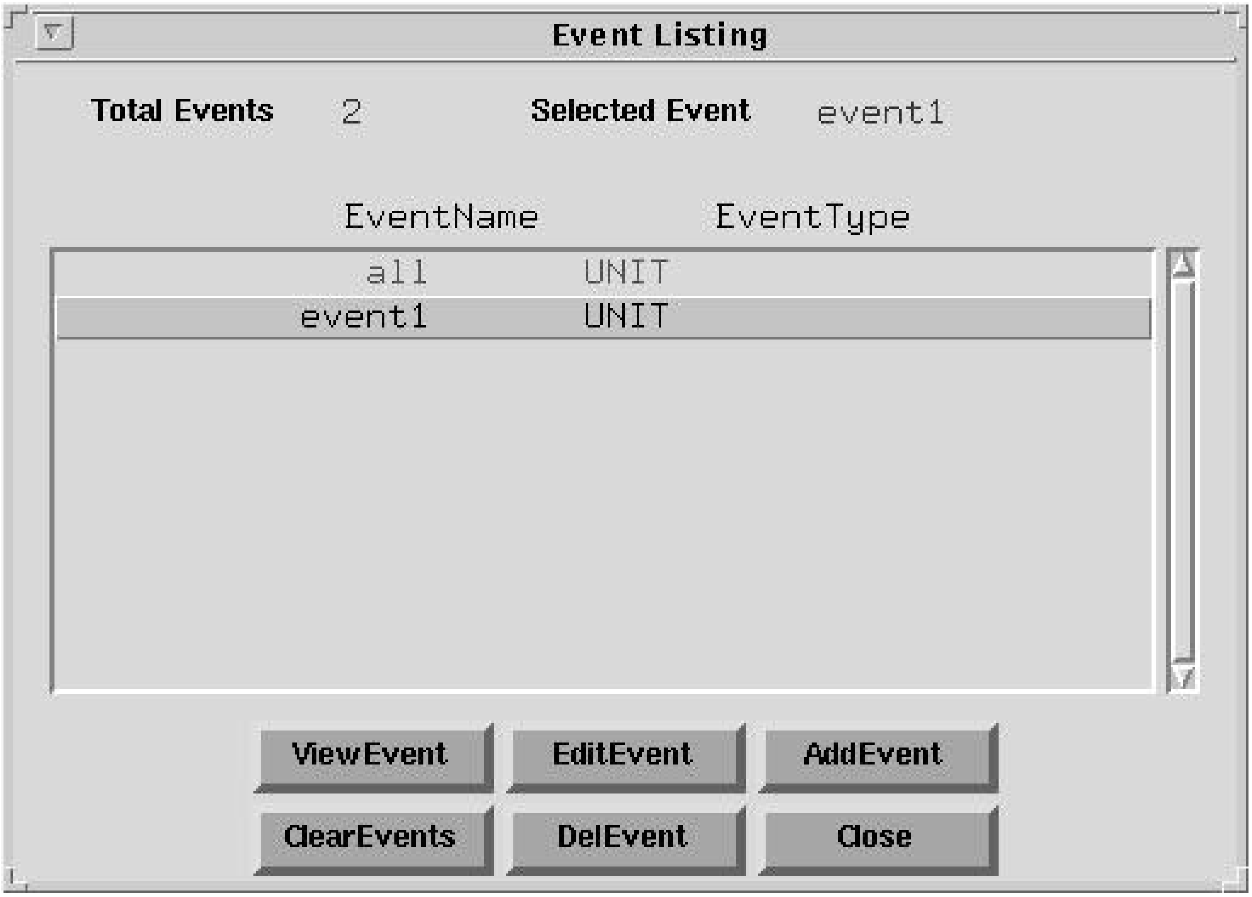

Figure 3-35. Event Listing CmdBlk Menu Panel

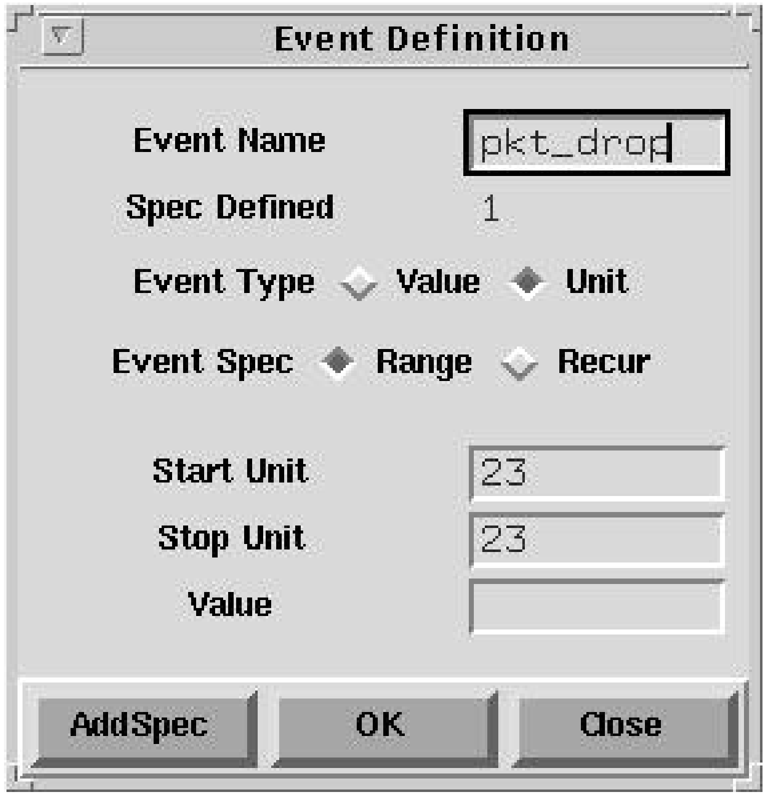

Figure 3-36a. Event Definition Menu Panels

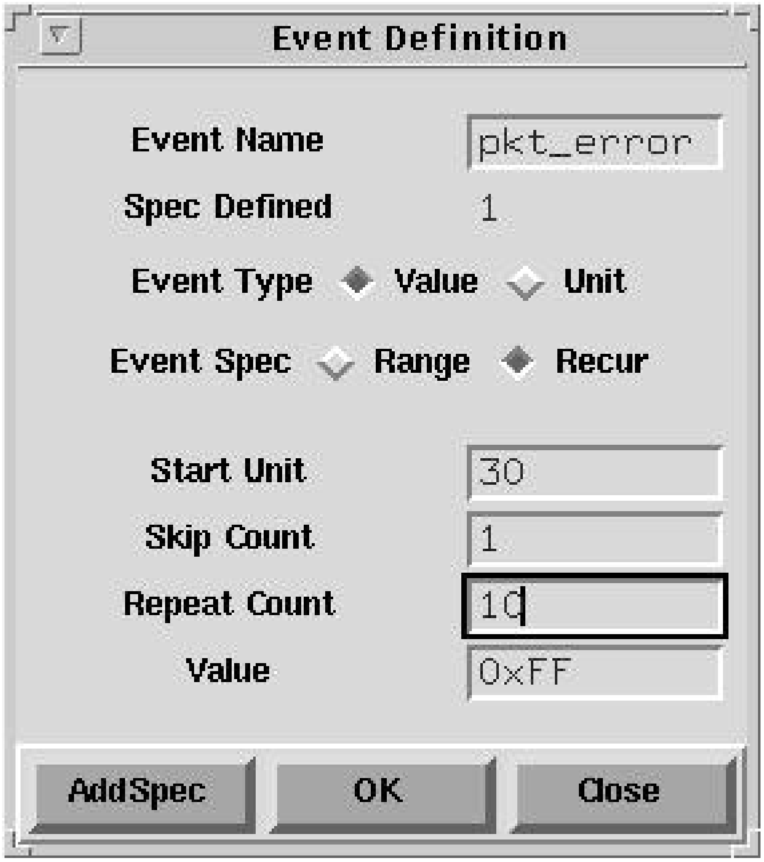

Figure 3-36b. Event Definition Menu Panels (Cont'd)

The next menu entry validates the name of the stream as 'Physical', and accepts the name of the instream, which is the next block on the heirarchy, i.e., 'cltu'. The maximum number of units is user selectable. The option to include the statistics in an expected results file is selected by using the 'Yes' option for ER file; and the 'Drop Units' option is selected by depressing the 'Yes' button. When the 'Drop Units' option is selected, the 'Drop Event' has to be selected on the 'Event Listing' menu (similar to that shown in Figure 3-35) which will be displayed. The definition of the 'Event' is similar to that described for the 'CmdBlk' Definition.

The user entries are saved by depressing the OK button and CLOSE will return the user back to the scenario menu shown in Figure 3-29.

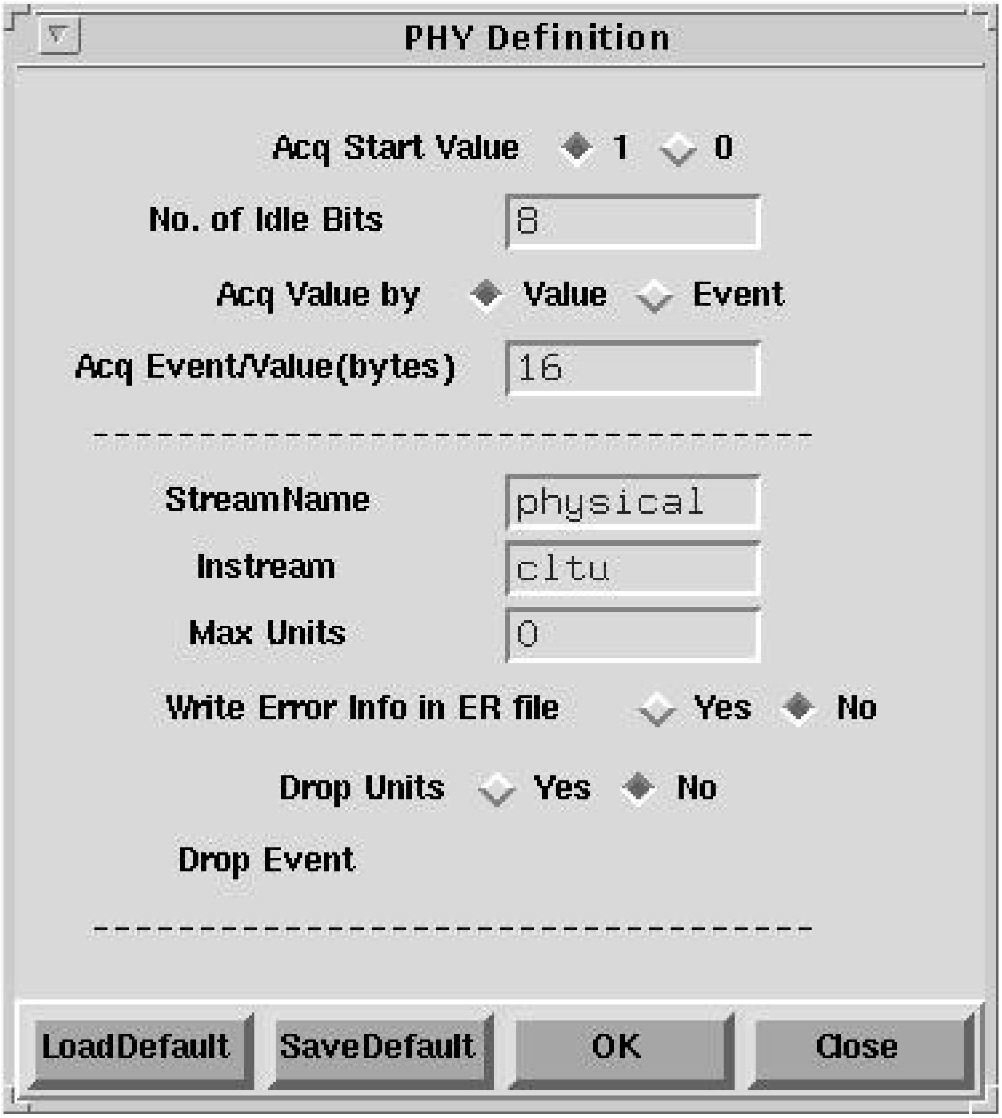

Figure 3-37. Physical Definition Menu Panel

The next sequential step in the scenario is the defining the contents of the 'Physical' stream, namely the CLTUs. Selecting this block on the TC Data Scenario menu will display the CLTU Definiton Menu Panel, as shown in Figure 3-38.

The first entry in this menu is the 'Code Block Length' specification in bytes. The EOS-AM1 Interface Control Document, Number 106, defines the code-block length as 6 octets (8-bit bytes). This will also append a 6-byte Tail Sequence in the CLTU generation. The sample script at the end of this section shows the format of the CLTU. The instream is predefined as a stream of 'cltu', and the Maximum number of Units is user-selectable. The option to include this information in the Expected Results or ER file and the option to 'Drop Units' specified by their unit index or sequence number is once again available to the user. The Event Listing Menu, similar to that shown in Figure 3-35, is also displayed at this hierarchical level to label and define the units to be dropped.

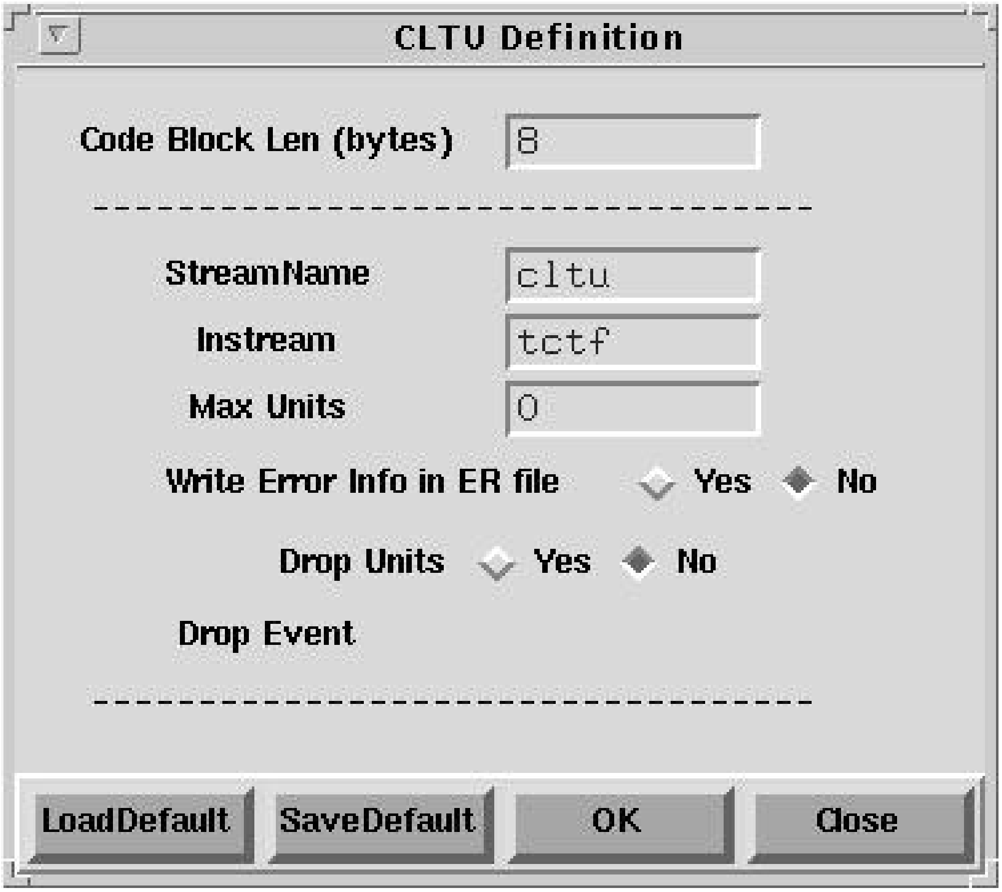

Figure 3-38. CLTU Definition Menu Panel

Once all the required information is entered, the OK button will save the user input and the CLOSE option will take the user back to the Telecommand (TC) Data Scenario menu panel depicted in Figure 3-29.

The next step in the scenario definition is to define the format and content of the TCTF or Telecommand Transfer Frame. The first three entries in the menu, shown in Figure 3-39, will define the Spacecraft Identification (SCID), the Virtual Channel Identification (VCID) and the Telecommand Transfer Frame (TCTF) length in bytes. The length field may be left blank, since the TCTF 'length' is dependent on the size of the packet, which has not yet been defined. The TCTF length is equal to the packet length plus the TCTF header length of 5 bytes and the optional 2 bytes if CRC encoding is selected. CRC encoding is selected by choosing the 'Yes' option.

Note: the TCTF 'length' includes the coding check symbols. Thus, the actual data in the TCTF is coded to a 6 octal code block which consists of 5 bytes of data plus 1 byte encoding chech symbol. The EDOS ground message header calculation at the end of this section shows an example of how this is generated.

The stream is validated if 'tctf' appears in the entry box, and the instream is selected as a Packet Identifier APID, i.e., ap001. Once again the maximum units provides the user some control on how many packets are to be created, and since there is one packet per frame selected for this option, the number should be the same, i.e., 100. The ER option and the 'Drop Units' options are set up in the same sequence of steps, as described in the earlier paragraphs for 'CmdBlk/Record', 'Physical', and 'CLTU'. This menu will also allow the user to optionally select the creation of the Expected Results (ER) file, which will by default have the name of the data scenario with a different extension, i.e., 'scenariofilename.er' Once the TCTF is defined, the OK button will save the entries, and the CLOSE button will return the user to the TC Stream Graphical Menu, Figure 3-29.

The first step in the process of defining the packet stream is to enter the Packet Identification. By selecting the Packet button on the TC Stream Menu, the Packet Identification Menu Panel, Figure 3-40, namely the 'Input Window', is displayed. The Packet is identified by writing in the APID number, i.e., 'xxxx'. Selecting OK will display the Packet Definition Menu shown in Figure 3-41. This menu is used to define the packets within the Packet/APID stream. The first four entries in this panel are already entered. The 'Max' entry is used to define the max number of packets that could be created from this APID stream. This is especially important when reading from a raw graphics file, in which the size is fixed. The option to create Telecommand packets is selected for forward link data. The Packet Data has to be defined, and by selecting the 'Yes' option for Packet Data, the Data Definition Menu Panel as shown in Figure 3-42a is displayed.

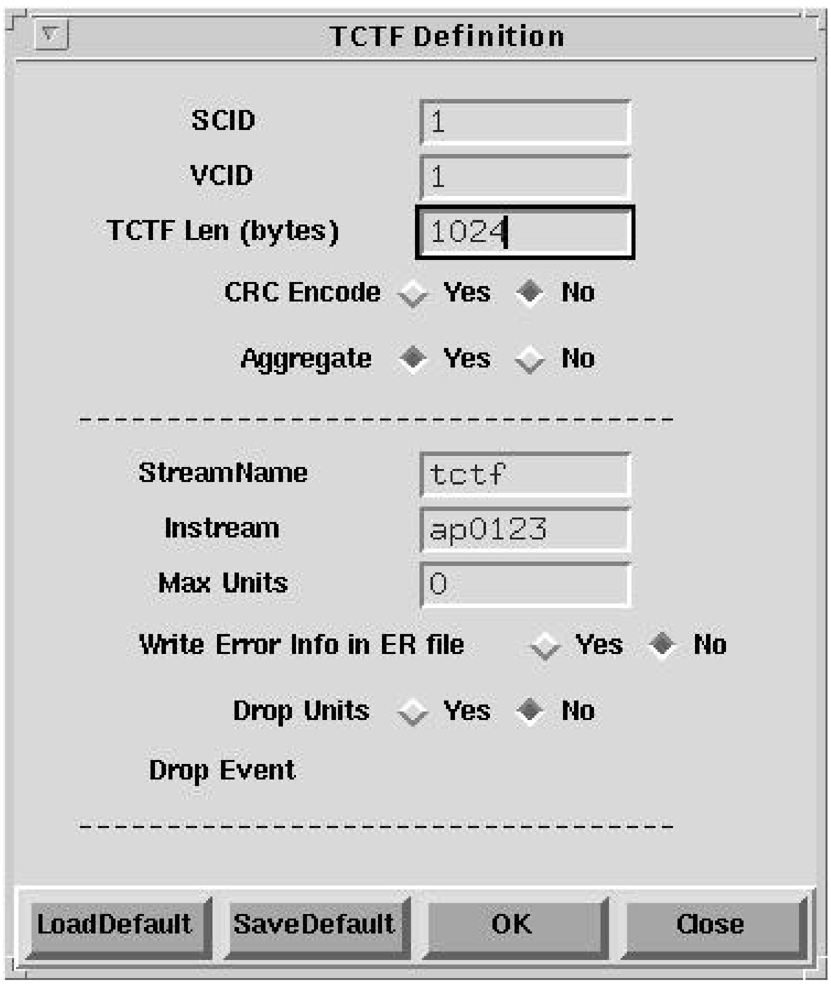

Figure 3-39. TCTF Definition Menu Panel

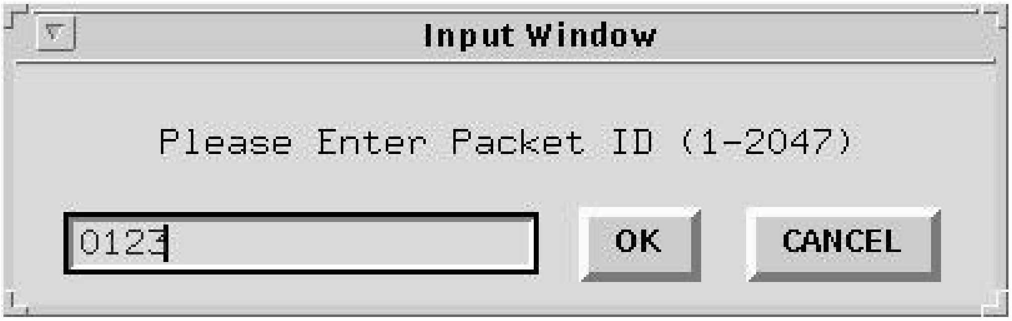

Figure 3-40. Packet Identification Menu Panel

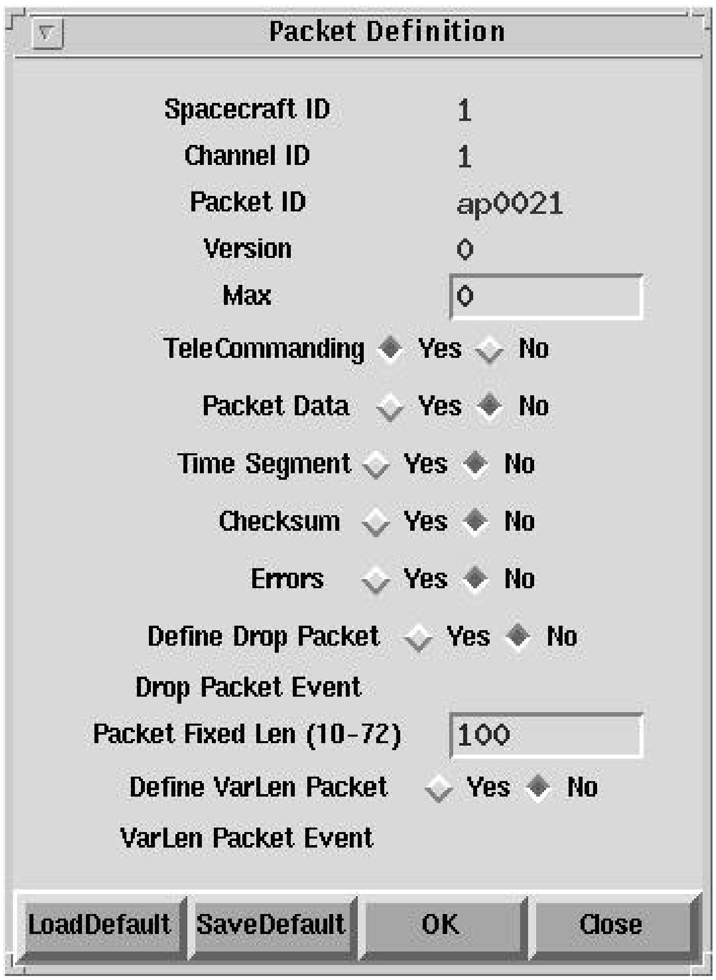

Figure 3-41. Packet Definition Menu Panel

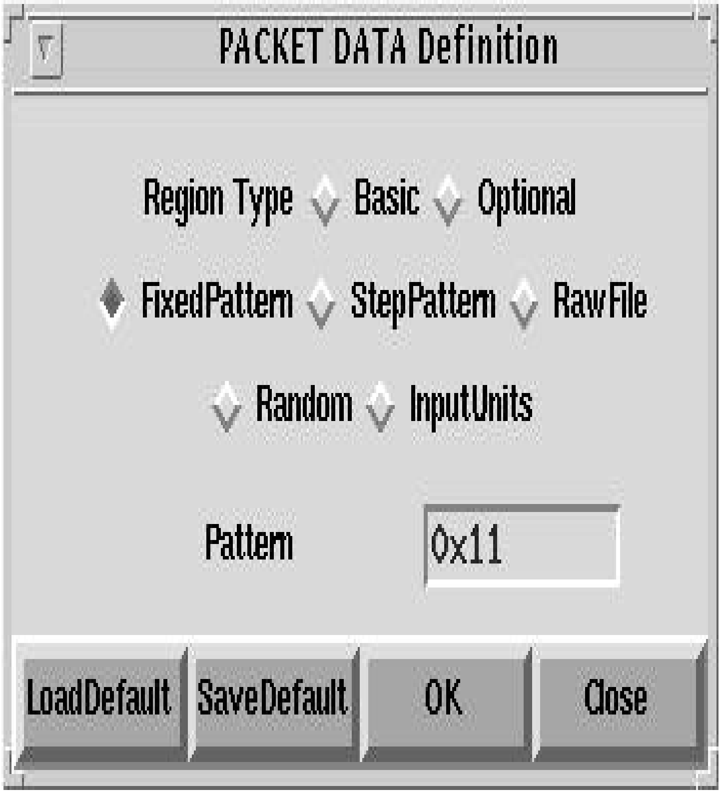

Figure 3-42a. Data Definition Menu Panel

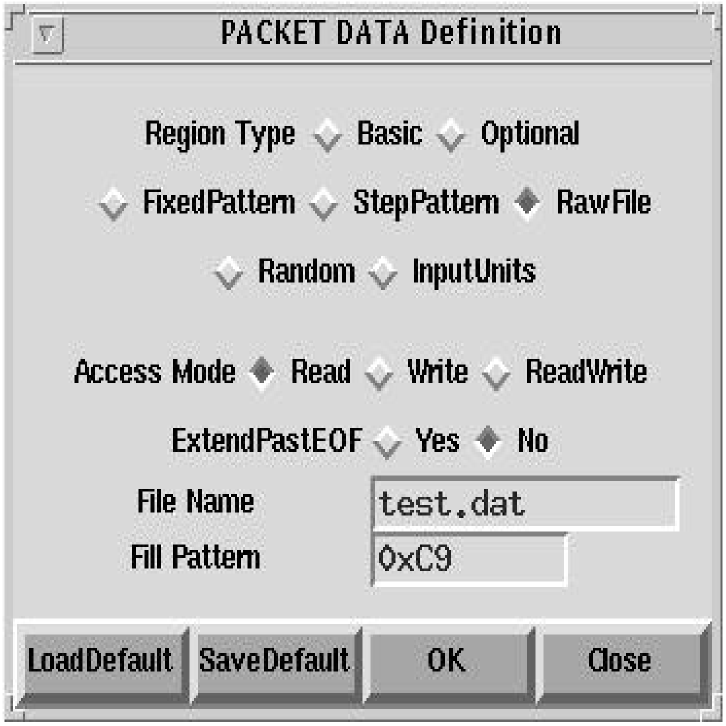

Figure 3-42b. Data Definition Menu Panel (Cont'd)

The contents of the data region can be either of a 'Fixed' pattern; 'Step' pattern; 'Raw File' or a 'Random' pattern. When the 'Raw File' option is selected, the menu panel shown in Figure 3-42b is displayed, where the name and path of the file have to be entered. In the 'Fixed' and 'Step' options the values have to be entered. OK and subsequently CLOSE, will save the input and return the user to Figure 3-41. For Forward Link data, there is no secondary header, thus the Second Header 'No' option is selected.

If the variable length option is selected the Event Panel, shown in Figure 3-45, will be displayed. The packet length, and the 'Event' that specifies the variation have to be defined. The 'Event' specification is similar to the 'Event' specification for inserting errors and drop units. OK will save the values and return the user back to Figure 3-41.

When the 'No' option is selected for 'Define Variable Length', a value for the 'Packet Fixed Length' has to be entered by the user. The 'Checksum' option is either selected or not, and when selected this performs a checksum on each packet and inserts the checksum value as the last byte in the packet data region. The Error' insertion process follows the same steps used for the error insertion in the Return-Link section for the packet level, and the steps are repeated here for completeness. Selecting the 'Yes' option on the next entry in the Packet Definition, will display the Error Listing Menu Panel, shown in Figure 3-43.

Any number of errors maybe introduced for the selected APID. In the Error Listing Menu, the AddError is depressed to display the default Error Definition menu panel, shown in Figure 3-44, which comes up for the 'Flip' error type. In this menu, the user can label the type of error; select the method by which the error is to be inserted, i.e., by flipping bits, or setting bits to specific values, or by completely dropping packets identified by the APID index or count.

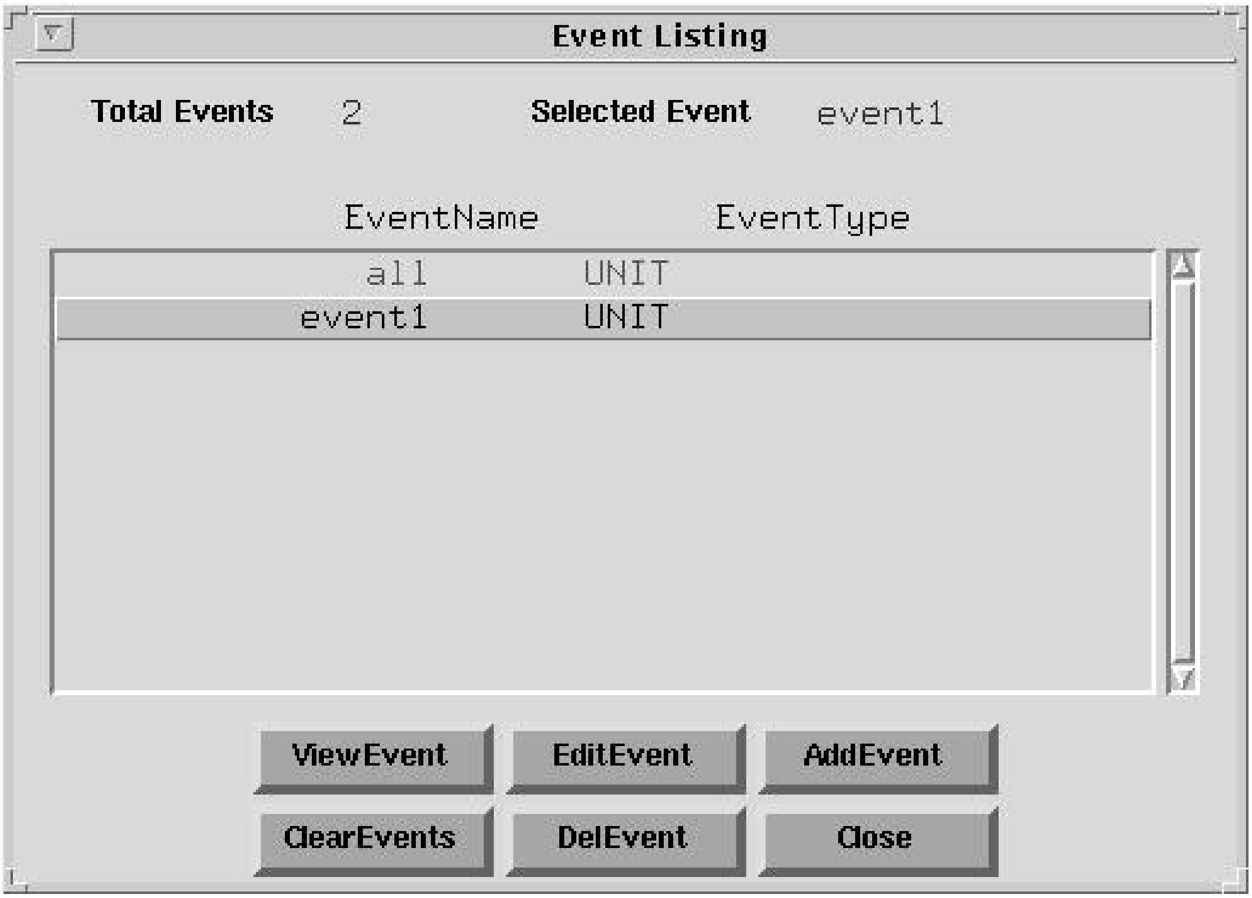

Figure 3-43. Error Listing Menu Panel

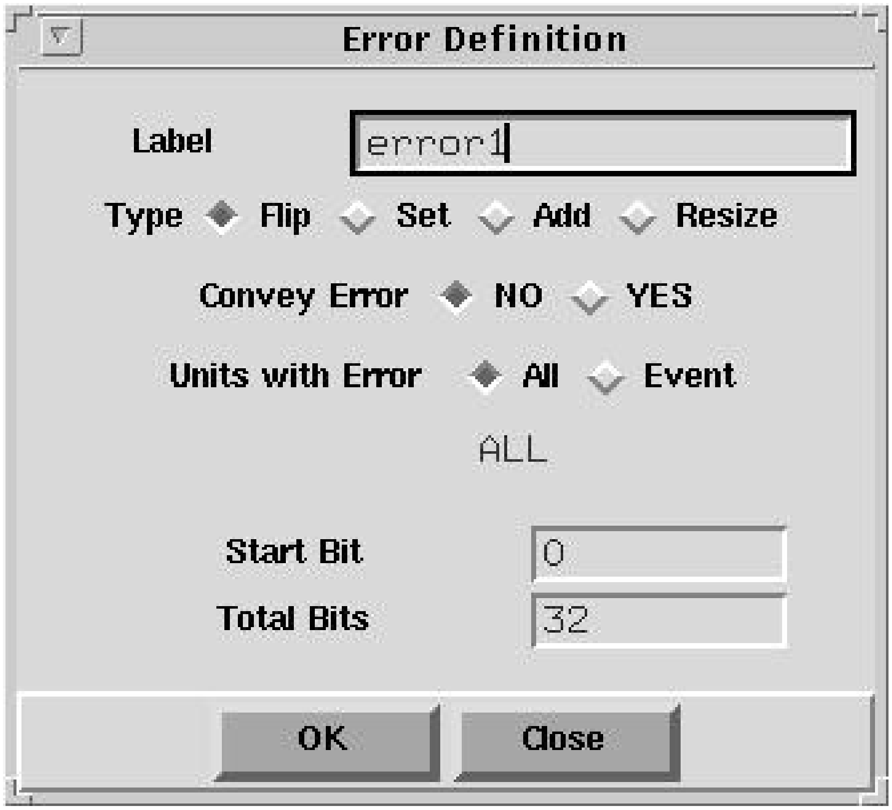

Figure 3-44. Flip Error Definition Menu Panel

When bits are to be flipped, Figure 3-45, the exact location of the bits within the packet, and the number of bits to be flipped are specified. The user can also specify whether this error event is to be implemented on all the packets from this APID stream or certain packets or ranges of packets as specified by their index in the stream. If all the packets are selected to have the error then the 'All' option is selected. If specific units are ear-marked for error insertion, then the 'Event' option is selected which displays the Event Listing Menu Panel, shown in Figure 3-45.

The AddEvent option displays the Event Definition Menu Panels, shown in Figure 3-46a, that allow the user to select the packets for error insertion. The Event name is user selectable and the occurrence of the event can be either a value or a unit. The Event Spec defines whether a certain range of units or packets specified by their unit number in the stream, Figure 3-46a; or a recurrent pattern is specified based on the start packet number; how many contiguous packets from this stream; how many packets are to be skipped; and how many times this pattern is to be repeated, Figure 3-46b. The OK and CLOSE options will save the selected input and return the user to the Error Definition Menu Panel.

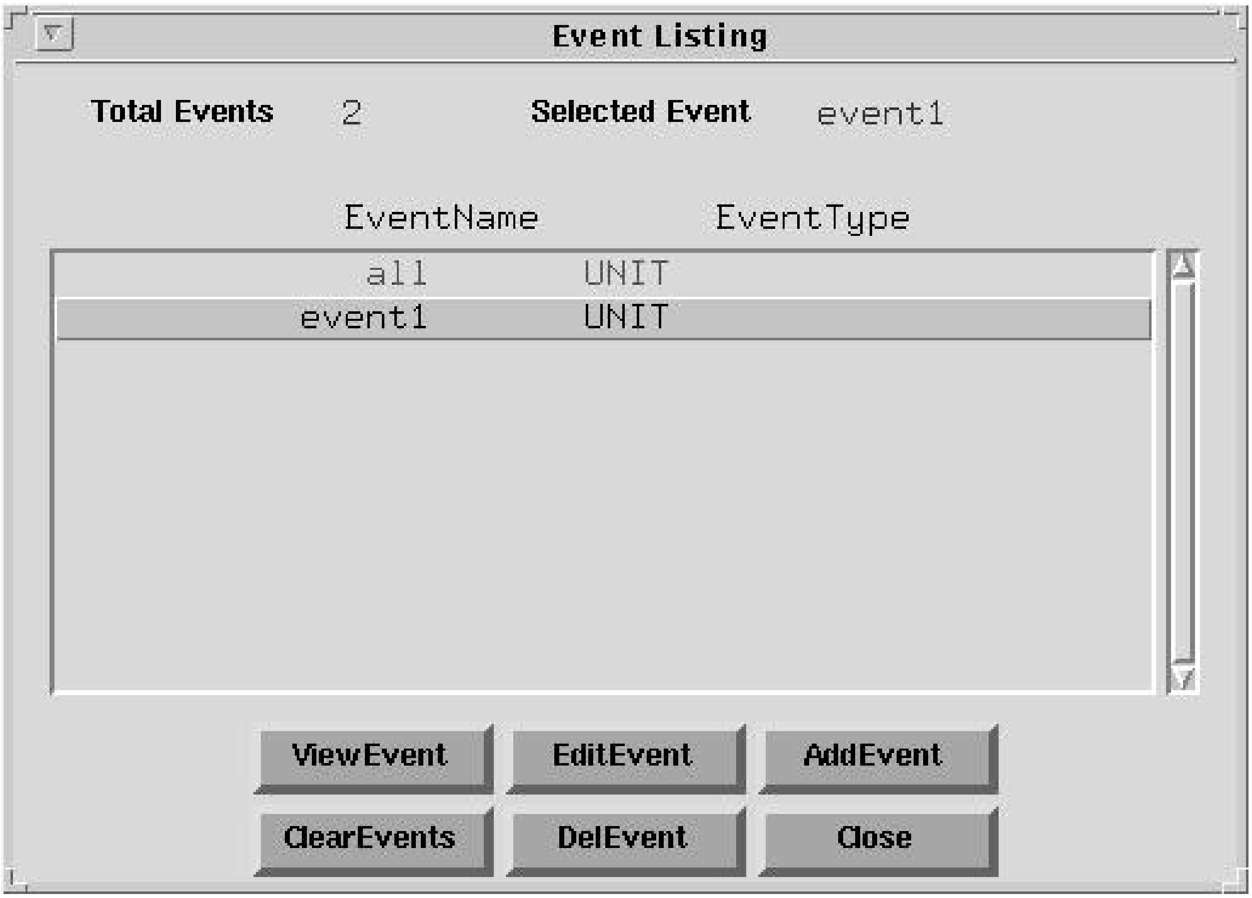

Figure 3-45. Event Listing Menu Panel

Similarly when bits are set to certain values, the user can select the label, occurrence, location and quantity of the bits and the value to be set, Figure 3-47. In dropping Packets, the user can select packets by their index or unit number in the VCID stream. The Convey function is optionally set to convey the error up to the frame encoding process. If this is not set, the frame encoding process will correct the errors inserted in the Packet, and the created test data will not contain the Packet level errors. The other buttons shown in Figure 3-43 are self-explanatory. The DelError button allows the user to selectively delete an error entry, and finally the Clear Error button clears out all the listed errors. OK and subsequently CLOSE, will save the input and return the user to the Packet Definition Menu Panel shown in Figure 3-41. Depressing OK once more will return the user to the TC Stream Graphical display shown in Figure 3-29.

At each stage in the hierarchy, when the user has completed entering the values needed for data generation and saved (or 'OK'ed the same), the box changes color from blue to green. When the box is selected it changes color from blue to yellow. Once the whole data scenario has been entered, the user can select the 'Script' option, whereby the Script Window is displayed, as shown in Figure 3-48a.

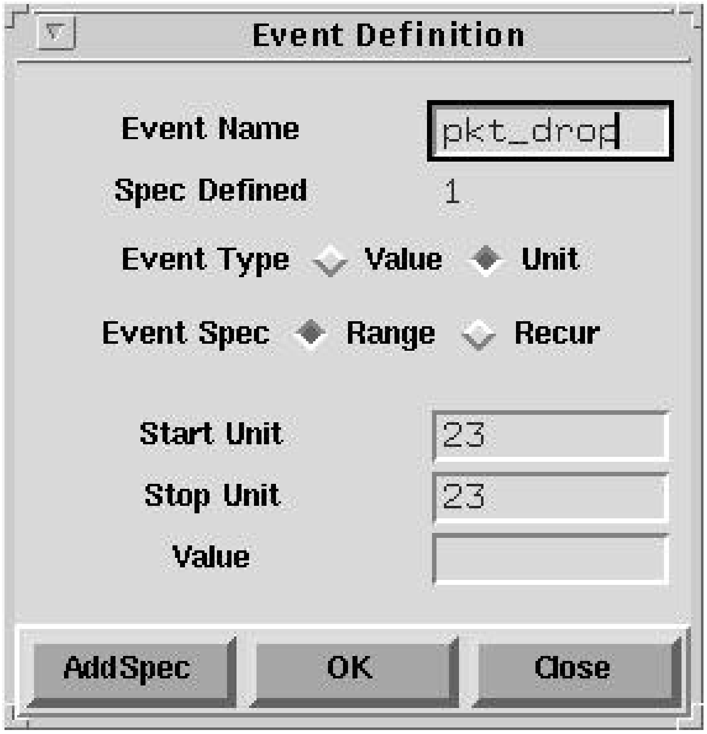

Figure 3-46a. Event Definition Menu Panels

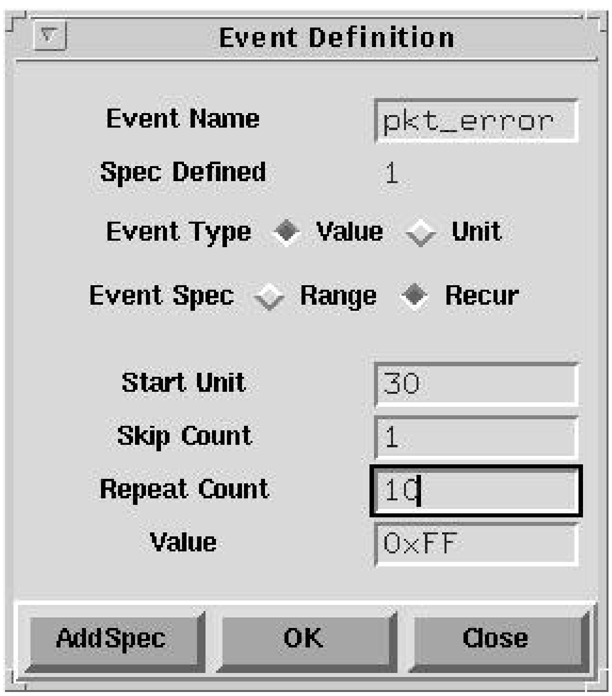

Figure 3-46b. Event Definition Menu Panels

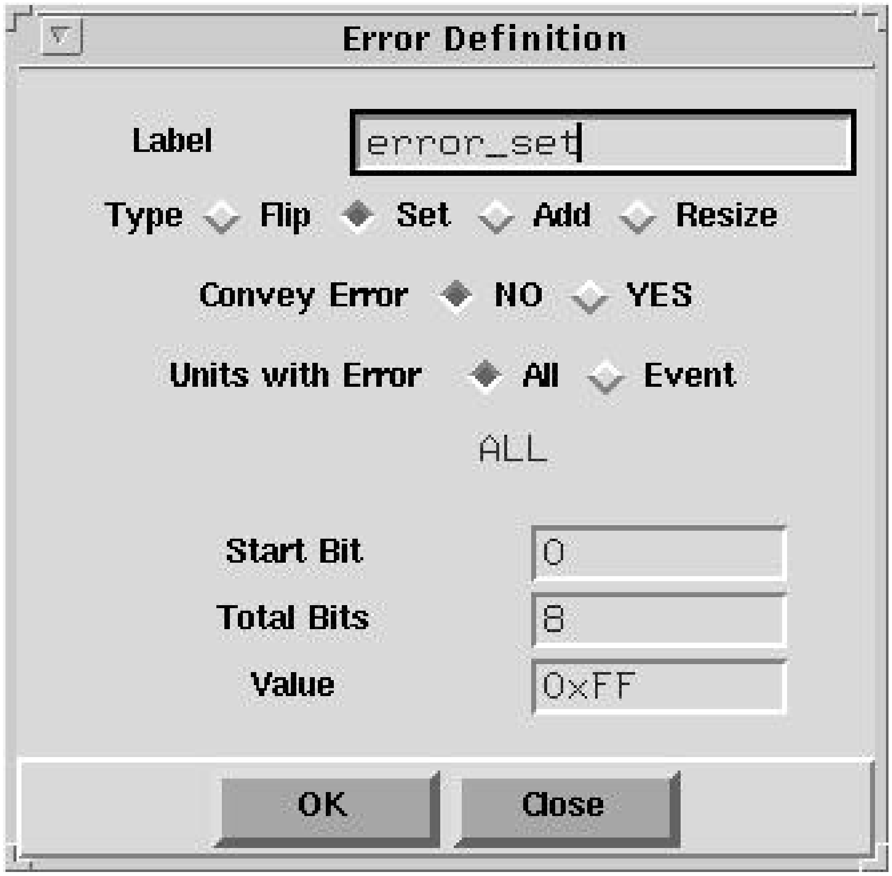

Figure 3-47. Set Error Definition Menu Panels

The script created has been commented to illustrate where the different argument values were input by the user and how they should appear in the script. The commented scripts are shown in Figure 3.48b. The description of the EDOS Ground Message Header is shown in Figure 3-49.

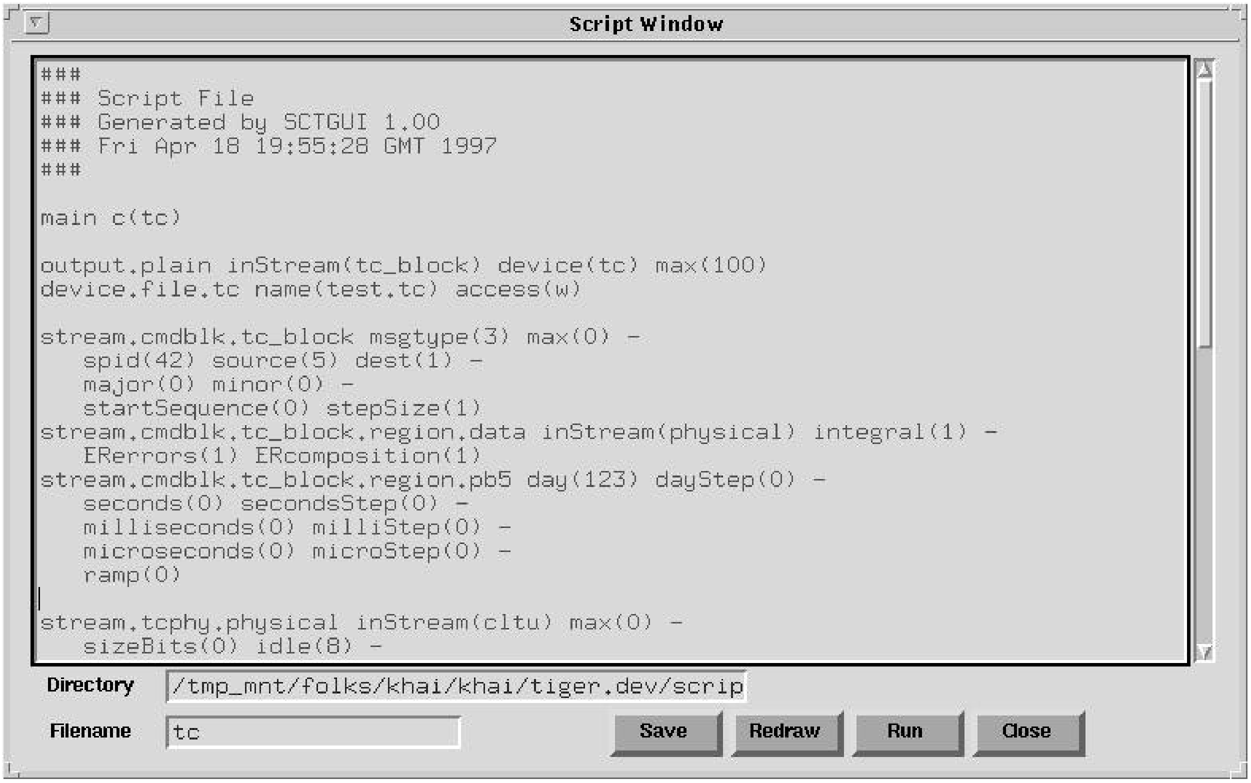

Figure 3-48a. Script Window for the CmdBlk

```
###
### Script File
### Generated by SCTGEN GUI 0.16
###
###
###
```

*** the first line main(tc) is created when the tc sceanrio is selected ***

```
       1. main c(tc)
```

*** the output definition menu creates the second line ***

```
       2. output.plain inStream(block) device(tc) max(100)
```

*** the device definition menu creates the third line ***

```
       3. device.file.tc name(cmtest3c.tcmd) access(w)
```

*** the command block definition menu and within this menu the PB5 time stamp definition menu create the fourth, fifth and sixth lines ***

```
       4. stream.cmdblk.block msgtype(3) source(5) destination(1) spid(42) -
              major(0) minor(0) max(0) startSequence(0)
       5. stream.cmdblk.block.region.data inStream(physical) integral(1)
       6. stream.cmdblk.block.region.pb5 day(445) dayStep(0) seconds(0) -
              milliseconds(0) microseconds(0) secondsStep(0) milliStep(10) -
              microStep(0) ramp(0) drift(0) driftFreq(0)
```

*** the physical definition menu creates the seventh line ***

```
       7. stream.tcphy.physical inStream(cltu) max(100) -
              sizeBits(0) idle(8) acquisition(128)
```

*** the command link transfer unit definition menu creates the eightth line ***

```
       8. stream.cltu.cltu inStream(tctf) max(100) -
              codeblockLength(6) sizeBits(0)
```

*** the telecommand transfer frame definition menu for vc1 creates the ninth line ***9. stream.tctf.tctf inStream(ap0001) max(100) -

```
              version(0) spid(100) vcid(1) aggregate(0)

# PacketScript
```

*** the packet definition menu for ap0001 creates the tenth line ***

```
       10. stream.pkt.ap0001 appid(0001)  -
              version(0) length(60) tc(1) checksum(1)  -
              variableLength(0)

# RegionScriptPACKET
```

*** the packet definition menu for ap0001 and within this menu the data definition menu creates the eleventh line ***

```
       11. stream.pkt.ap0001.region.data type(F)  -
              pattern(0xa5)

# Event statements
```

*** no event statements were created ***

Figure 3-48b. Commented Script for the CmdBlk

```
03000501 00837A00           EDOS GMH (24)
00000000 002A0000
0000007F 00000000

AAAAAAAA AAAAAAAA    Acquisition (16)
AAAAAAAA AAAAAAAA

EB90                        Start Sequence (2)

0064044000 9C        TCTF Hdr. within codeblock (codeblock = 6)

1001C00000 96        Packet Hdr.(5/6). within codeblock
35A5A5A5A5 B0        Packet Hdr.(1/6) + Data within code block
A5A5A5A5A5 BA        Source Data within code block
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A5A5 BA               "
A5A5A5A54F 72        Source Data(5/6) + checksum within code block

55                          Idle (1)

55555555 5555        Tail Sequence (= codeblock length = 6)
```

Figure 3-49. EDOS Ground Message Header Calculation and Format

3.4 EODS DATA PRODUCTS

SCTGEN can also be used to create the EDOS Data Products. These products are the Production Data Sets (PDS), Expedited Data Sets (EDS), Rate Buffered Files (RBF), and EDOS Data Units (EDU). The products use specific classes in the SCTGEN software and each one of them will be described in the following paragraphs.

3.4.1 PRODUCTION DATA SETS

The main SCTGEN GUI menu panel with the PDS option selected is shown in Figure 3-50. The process used to define a PDS is the same as for an EDS, which is described in Section 3.4.2. PDS files will be given the extension "PDS".

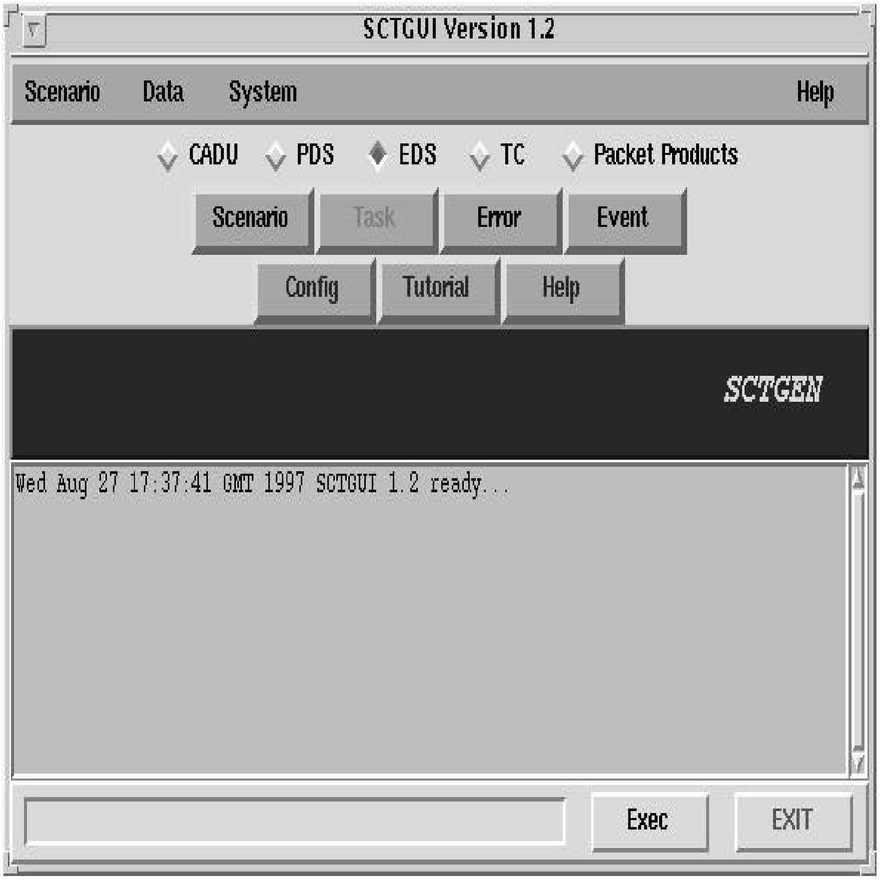

Figure 3-50. SCTGEN Main Menu Panel

3.4.2 EXPEDITED DATA SETS

The main SCTGEN GUI Menu Panel with the EDS option selected is shown in Figure 3-51. Selecting the menu option scenario -> new will open the EDS Input Window, shown in Figure 3-52.

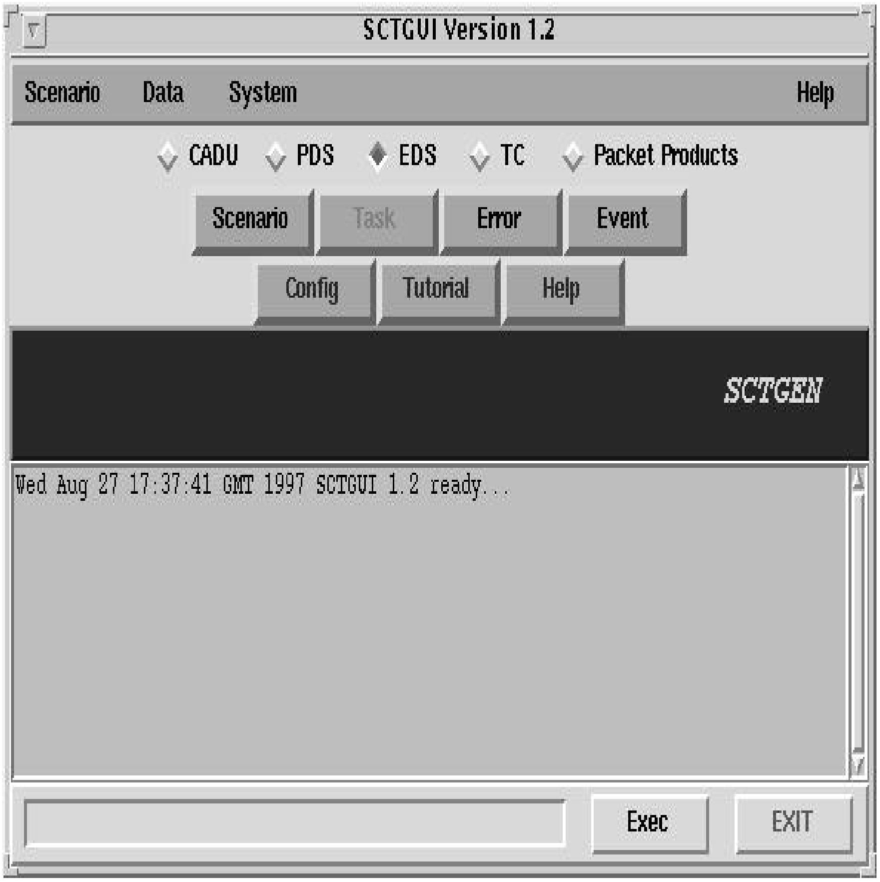

Figure 3-51. SCTGEN Main Menu Panel

This window allows the user to specify the Spacecraft ID. After entering the selection, OK will display the EDS Scenario Definition Menu Panel, shown in Figure 3-53.

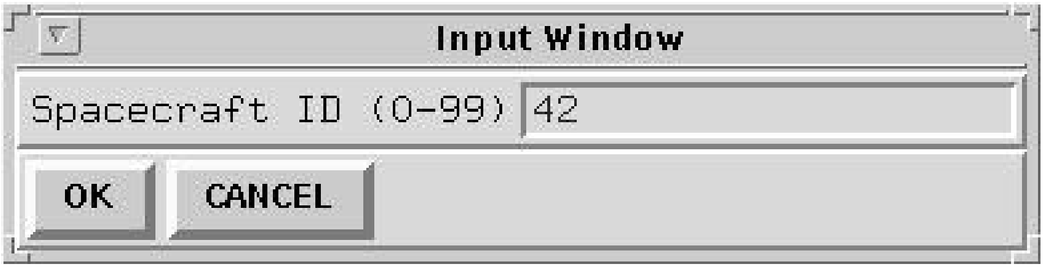

Figure 3-52. EDS Input Window Definition Menu Panel

This menu enables the user to enter the Spacecraft Identifier, VCID, APID and respective packet length for the EDS to be created. When all the entries are filled in, the OK button saves the input and VIEW takes the user to the EDS Stream graphical menu, shown in Figure 3-54. Each box in the heirarchy is defined by selecting the box and displaying the definition menus.

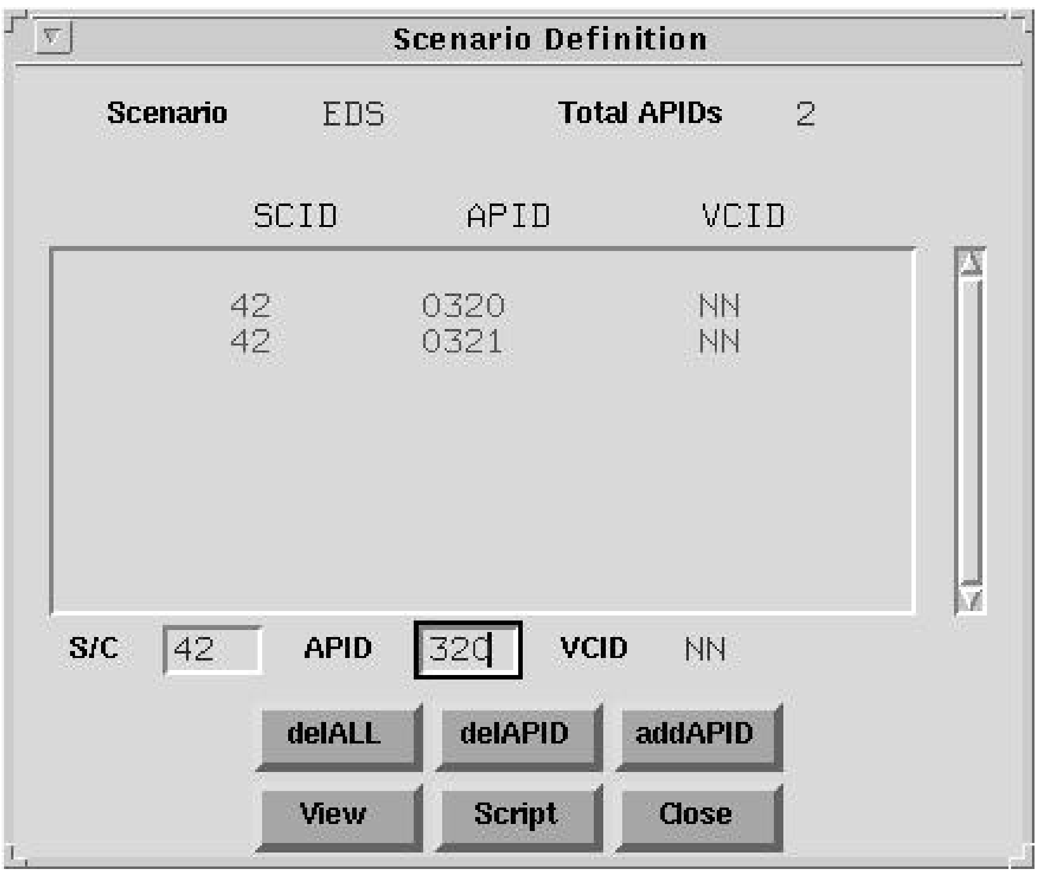

Figure 3-53. EDS Scenario Definition Menu Panel

The first box that is selected displays the Output Definition Menu Panel, shown in Figure 3-55. The Output is defined such that the construction record is created with the EDS. The first selection is the output type, and for EDS, this button is selected. The other fields are entered in with reference to the EDOS-EGS ICD. The first field specifies the size of each data set file within the EDS.

Thus, if the total size of the Data Set is 12700 KB, the user could elect to have 2 smaller files, each of 6350 KB. The VCID and SPID are entered in automatically from the EDS Scenario Definition entries. The major and minor software versions are user-selectable. The Data Set counter is a part of the Construction Record, and the range is from 0-9. The default value of 0, ensures that the first file is numbered from 0. The next field validates the output stream name and the instream in this case is a vcNNmux stream, which can input up to three different streams of packets from the Packet Producers. For the example described, the vcNNmux will be fed by two APID streams 'ap0320' and 'ap0321'.

The 'Max Units' restricts the number of actual packets to 1000. The Major and Minor entries are user selectable for specifying the software version.

Note: This version of the software needs to have a non-zero value entered for file size.

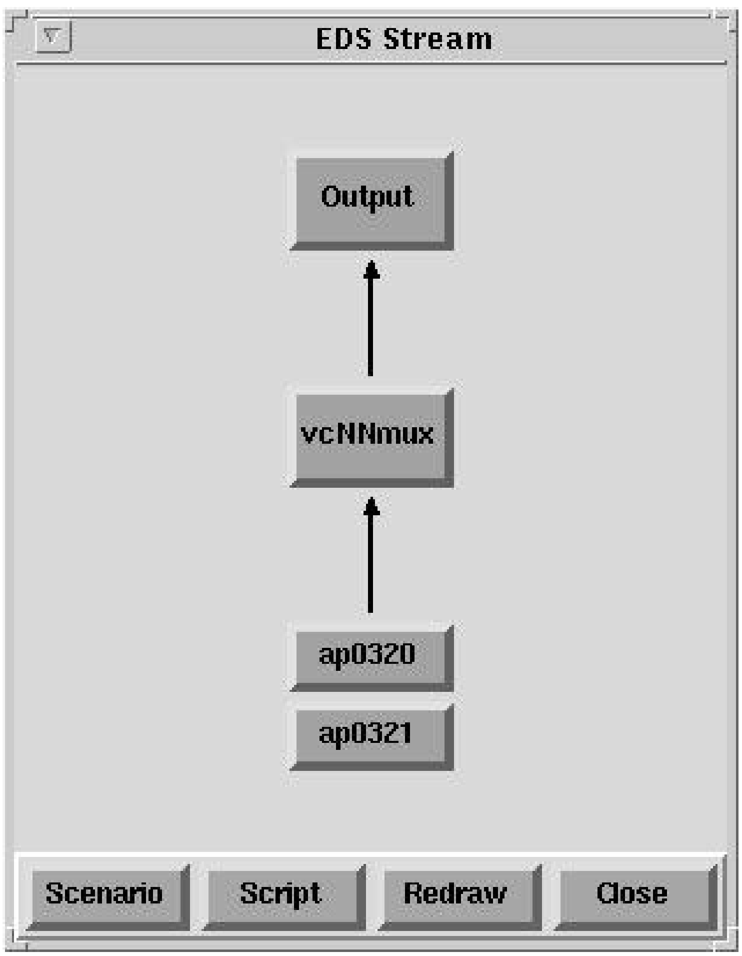

Figure 3-54. EDS Stream Definition Menu Panel

Once the fixed values are input, the time-stamps have to be initialized. The Creation Date when selected will open the Creation Definition Menu Panel, shown in Figure 3-56. The create date follows the convention “yydddhhmmss”, which is d=days, h=hours, m=minutes, and s=seconds. OK saves the input and returns the user back to the Output Definition Menu Panel, shown in Figure 3-55. Selecting the SC Time, displays the SC Time Definition Menu Panel, shown in Figure 3-57. This menu is easier to fill as it only requires the day, seconds of the day; milliseconds of the day, and microseconds of the day. OK and CLOSE return the user to the Output Definition Menu Panel.

The ‘PB5 time’ is selected to initialize the time stamp in the ESH Header. Depressing the ‘Yes’ button, displays the ESH Definition Menu Panel, shown in Figure 3-58. A pair of times have to be entered, i.e.,, start time and stop time for each segment of data. The ‘Day’ entry defines the start day (Julian calendar) from a user selected epoch for the Packet stream within the PDS. The default that has been calculated and set is from October 10, 1995.

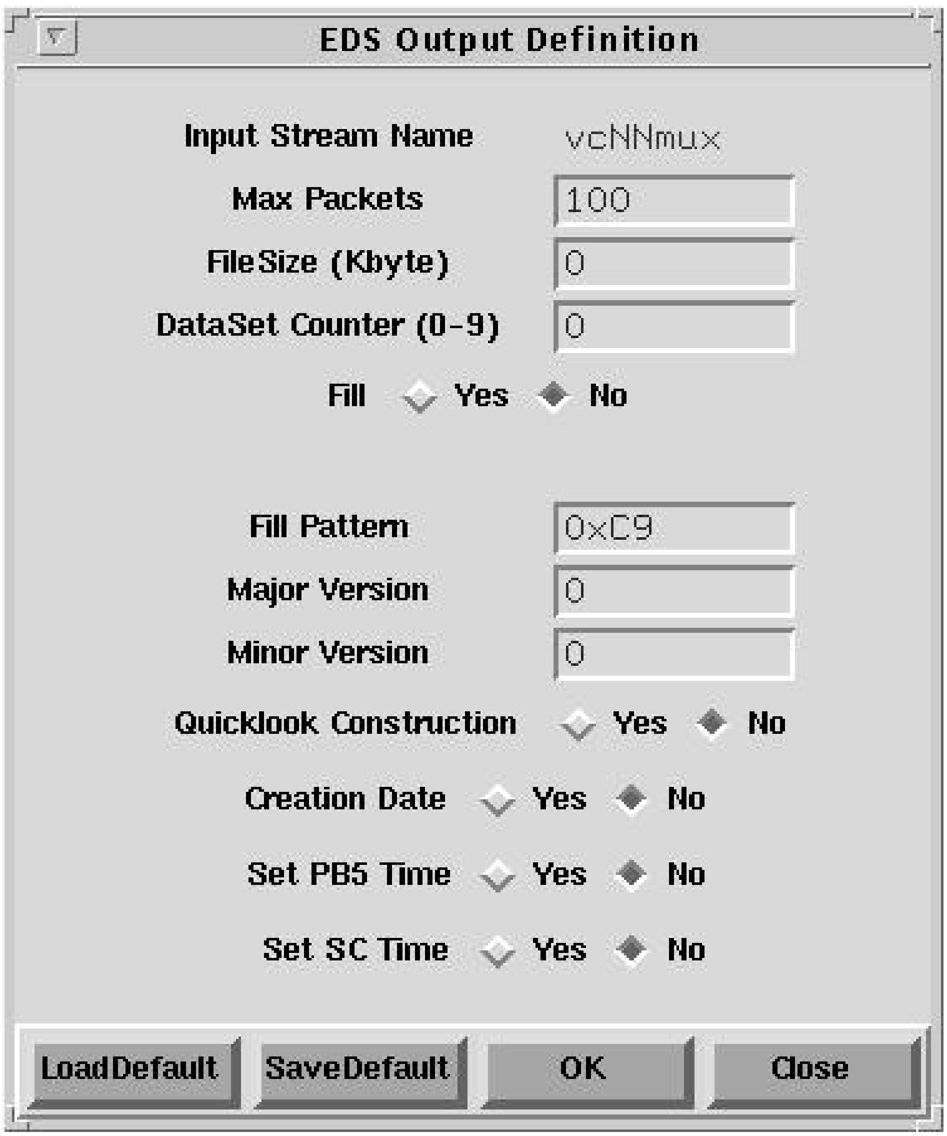

Figure 3-55. Output Definition Menu Panel

The 'DayStep' defines the day increment and the default is 1 day at a time. The 'Seconds' initializes the start time in seconds of the selected day, the 'Milliseconds' initializes the milliseconds and the 'Microseconds' initializes the microseconds of the selected day of the Packet stream time stamp. The next three entries, 'SecondsStep', 'MilliStep', and 'MicroStep' define the step in seconds, milliseconds and microseconds between consecutive packets within the EDS. The user is provided with the capability of defining a 'Ramp' function for the time stamp if required, and the correction due to Drift. When the 'Ramp' option is selected, the user is prompted to specify the number of units, in this case Packets, that will have the same time stamp, before the stime is stepped up to the next defined value.

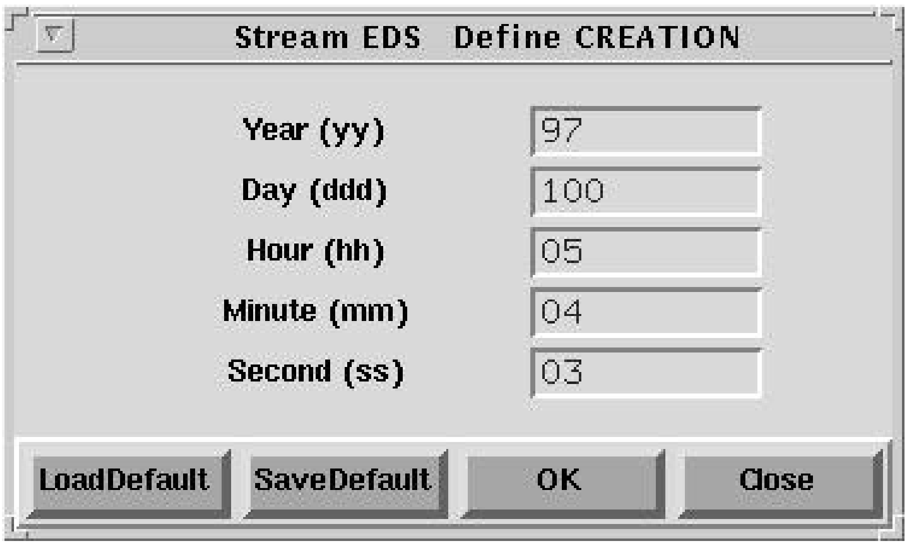

Figure 3-56. Creation Definition Menu Panel

The 'Drift' and 'Drift Frequency' allows the user to introduce a non-uniform step in time at selected units. OK saves the input and returns the user back to the Output Definition Menu Panel. Once all the entries in the Output Definition have been entered and saved, the CLOSE button will return the user back to the EDS Stream Menu shown in Figure 3-54.

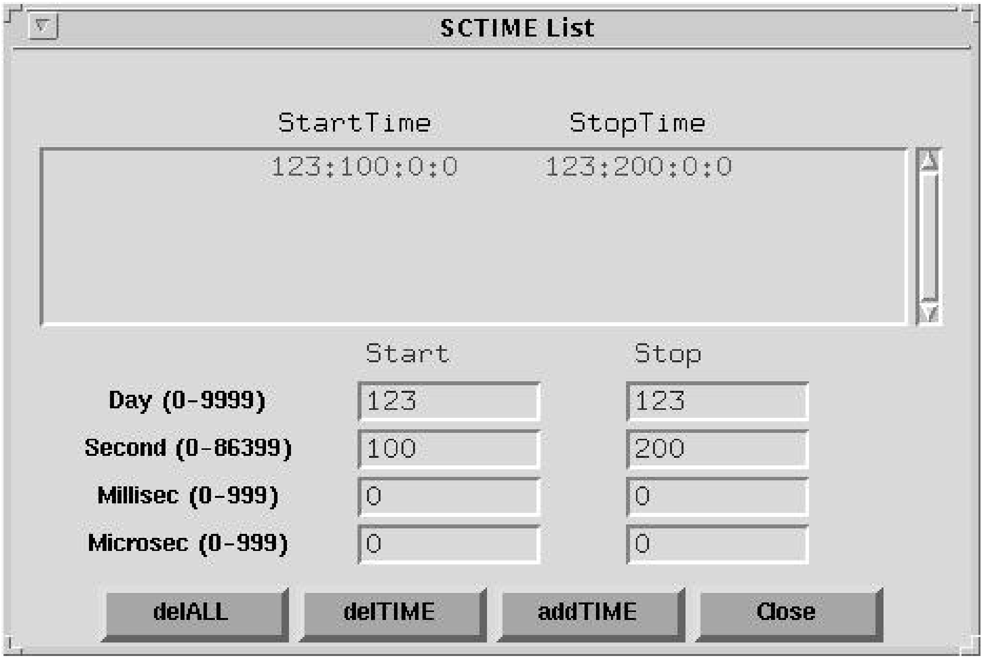

Figure 3-57. SC Time Definition Menu Panel

The next step in the definition is to define the multiplexing strategy for the APID streams within the EDS. Selecting the 'vcXXmux' button on the EDS Stream Menu Panel, will display the Packet Multiplexer Menu Panel shown in Figure 3-59. The commonest strategy will be to concatenate the APIDs serially, i.e., first the packet stream from APID 320 and then the packet stream from APID 321.

Finally, the last button on the EDS Stream is selected to define the Packet Stream. For each APID, a separate menu has to be opened. The Packet Definition Menu Panel for APID 320, shown in Figure 3-60, is the same as that displayed in Sections 3.3 and 3.4. However, in this case the Packet Data is defined for Return Link Data, so the Telecommand Option is not selected. When Packet Data is depressed, the Data Definition Menu Panel, (see Figure 3-42a) is displayed. The contents of the Packet Data are defined in the same way as they were defined for the PDS scenario described in the previous section. The 'Fixed Pattern' option allows the user to input a fixed pattern that will be repeated in the application data region of the packets. If the data is to be fed in from an external source, the device and path name have to be entered when the menu panel, under 'Raw File', is displayed. Once all the entries have been completed, the OK button saves the input and CLOSE returns the user to the Packet Definition Menu Panel. The 'Step' and 'Random' Pattern are described in the user reference and detail script guide included in later sections of this document.
The Spacecraft ID, Channel or VCID, and Packet Identifiers are pre-entered from the other menu panels. The secondary header option is selected as 'Yes' and the length defined. To define the time-code within the packet header, the Second Header option is selected. This displays the 2HDR Definition Menu panel.

For this application, SCTGEN has been customized for EOS-AM, and thus the tcEDOS option is used to initialize and define the step size for the time-code. The 'Day' entry defines the start day (Julian calendar) for the packet stream. The 'MS of Day' initializes the start time in milliseconds of the day when the packet stream should start, and 'MICRO of MS' initializes the micro-second granularity in the start time. The next two entries define the step in milliseconds and microseconds between consecutive packets within the APID or packet stream. The user is provided with the capability of defining a 'Ramp' function for the time code if required, and the correction due to Drift. The values that can be entered in these boxes have been described in earlier sections of this document. OK saves the user-input and returns the user back to the Packet Definition Menu Panel.

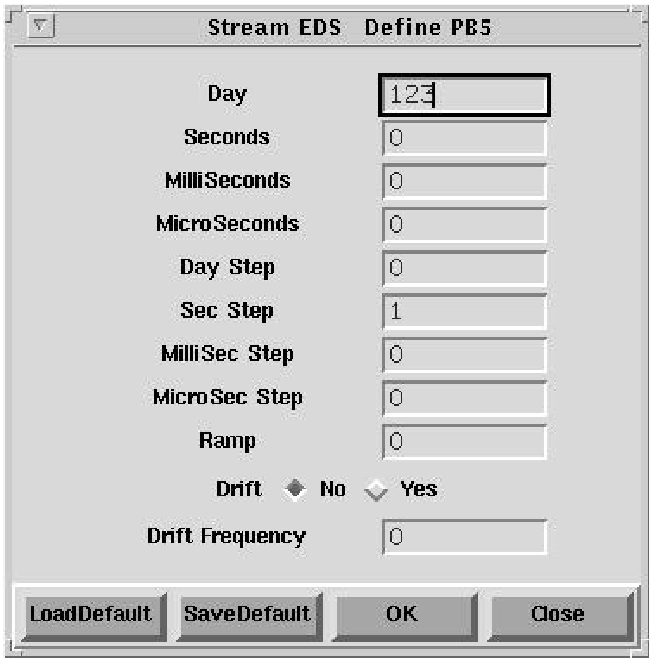

Figure 3-58. PB5 Time Stamp Definition Menu Panel

The packet length is selected as a fixed value, in accordance to the EDOS-EGS ICD, and the options to insert 'checksum', 'errors' and to 'drop' selected packets is implemented similar to the Return-Link Scenario. The 'drop' option when selected will display the Event Definition Menu Panel, shown in Figure 3-75. The 'unit' option specifies the packets that will be dropped, and the 'event name' will alert the Packet stream that these packets are to be dropped. The user must select 'Add Spec', before the OK and CLOSE sequence to register the event on the Packet Definition Menu display.

For an APID, any number of errors may be introduced. The error definition process is the same as for the PDS. The AddError is depressed to display the default Error Definition menu panel, similar to that shown in Figure 3-23, which comes up for the 'Flip' error type. In this menu, the user can label the type of error; select the method by which the error is to be inserted, i.e.,, by flipping bits, or setting bits to specific values, or by completely dropping packets identified by the APID index or count. Once the error is defined, the occurrence of this error has to be defined by specifying an event. If specific units are ear-marked for error insertion, then the 'Event' option is selected which displays the Event Listing Menu Panel, shown in Figure 3-62.

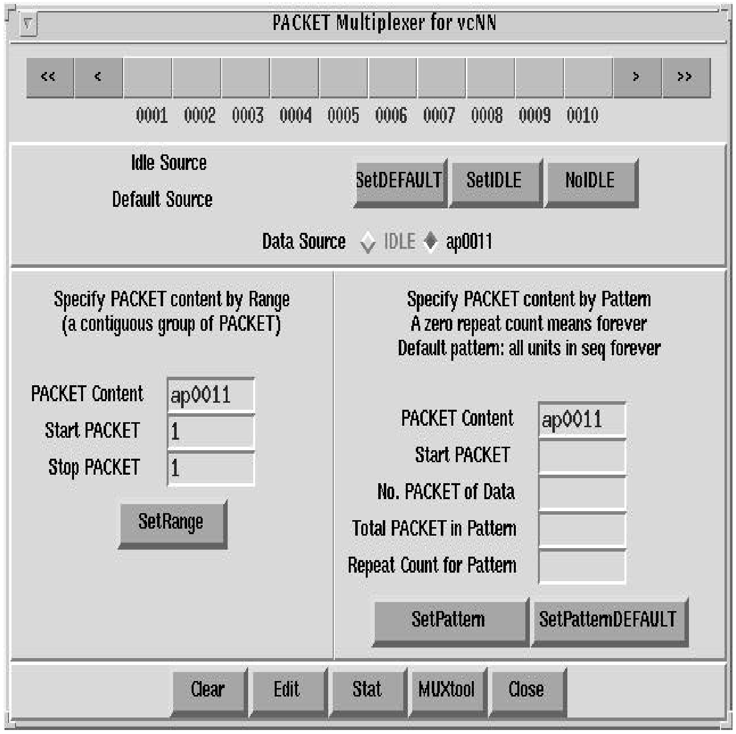

Figure 3-59. Packet Multiplexer Definition Menu Panel

The AddEvent option displays the Event Definition Menu Panels, described in the previous paragraph, that allow the user to select the packets for error insertion. The Event name is user selectable and the occurrence of the event can be either a value or a unit. The Event Spec defines whether a certain range of units or packets specified by their unit number in the stream, (see Figure 3-26a); or a recurrent pattern is specified based on the start packet number; how many contiguous packets from this stream; how many packets are to be skipped; and how many times this pattern is to be repeated, (see Figure 3-25b). The section on the script descriptions will provide examples of both options. Once the event is defined, the 'Add Spec' button MUST be depressed for the event to be registered, and then the OK and CLOSE buttons in that order will take the user back to the Event Listing Menu in Figure 3-62. The CLOSE options will save the selected input and return the user to the Error Definition Menu Panel. The Convey function is optionally set to convey the error up to the RS encoding process. If this is set, the RS encoding process will correct the errors inserted in the Packet, and the created test data will not contain the Packet level errors.
The user has to specify the Virtual Channel ID in the last part of the packet definition. The user can also specify how many packets have come from Reed-Solomon corrected frames. This will be an entry in the construction record for the EDS. Once all the entries in the Packet Definition Menu have been entered, the OK will save the input and CLOSE will return the user back to the EDS Stream Menu Panel. The same sequence of specifying has to be repeated for any other APIDs in this EDS. If there are two APIDs and hence the user repeats the Packet Definition for next APID.

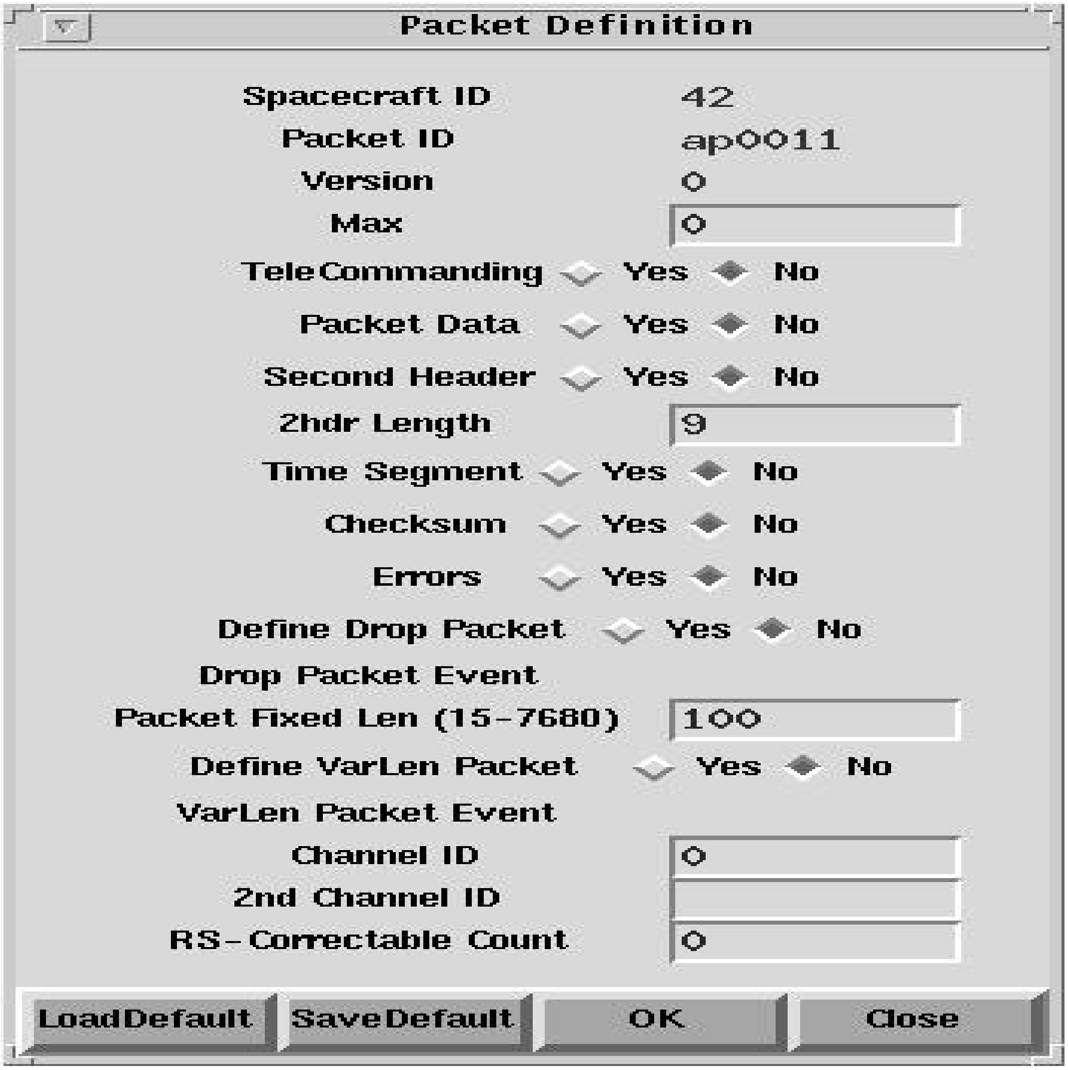

Figure 3-60. APID 320 Packet Definition Menu Panel

Finally, when all the fields have been defined, the OK and CLOSE returns the user back to the EDS Stream menu shown in Figure 3-53. When the 'Script' button is depressed the Script Window is displayed, shown in Figure 3-63. The 'Run' button will start the creation of the EDS and the Run Window, shown in Figure 3-64.

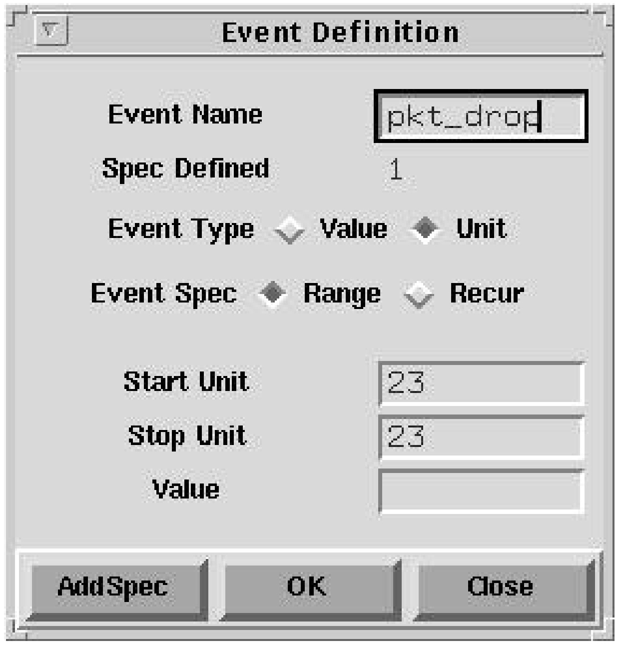

Figure 3-61. Event Definition Menu Panel

Figure 3-62. Event Listing Menu Panel

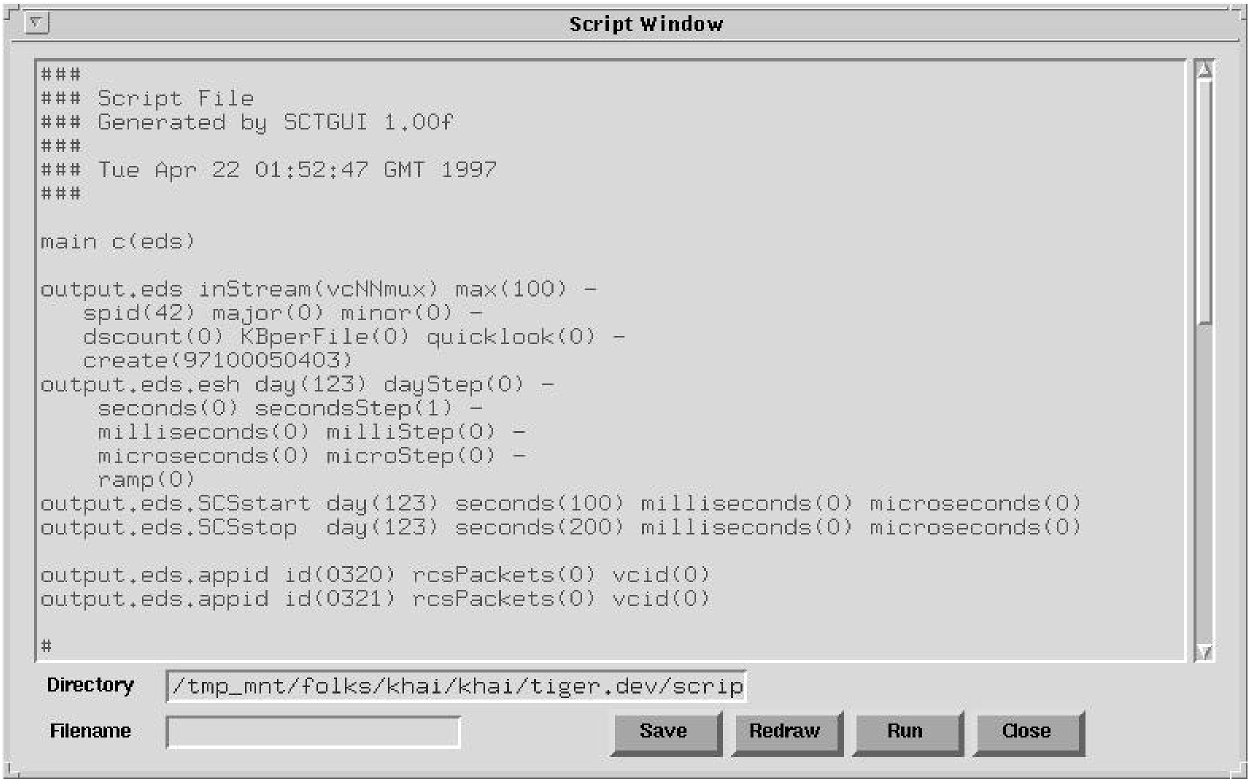

Figure 3-63. EDS Script Window

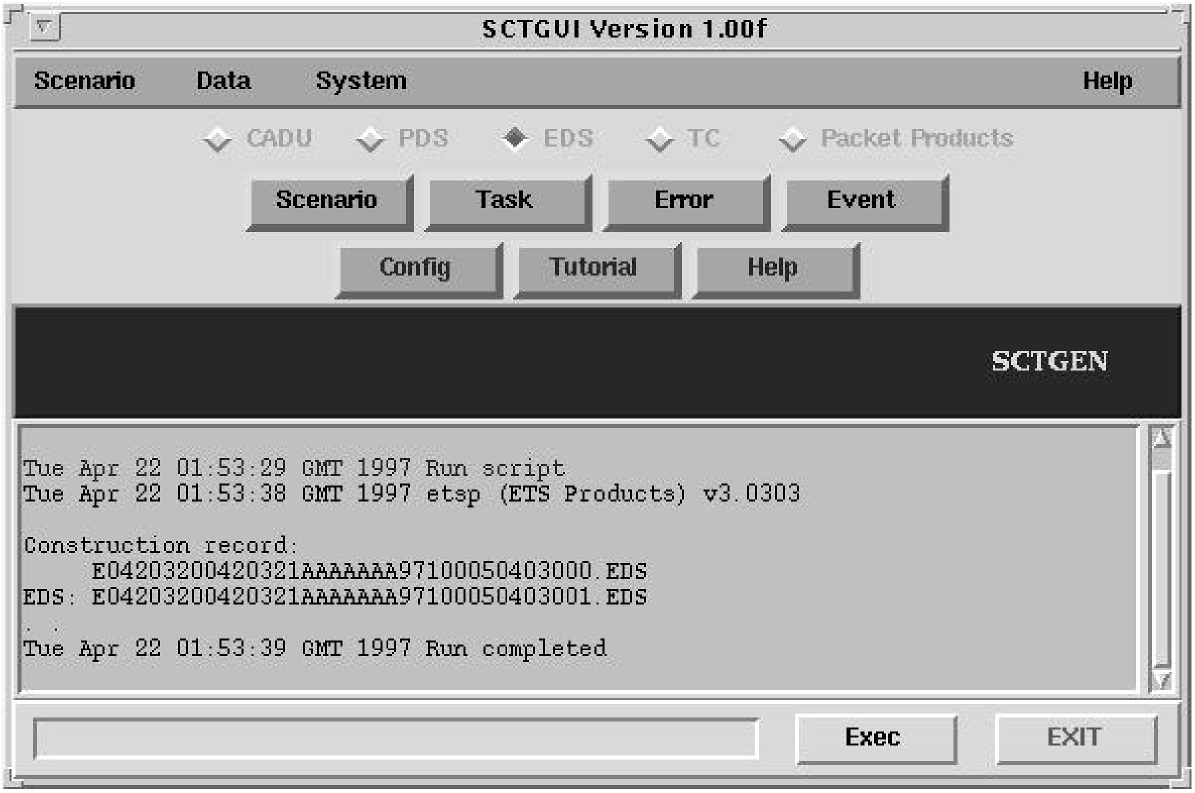

Figure 3-64. EDS Run Window

3.5 EDOS PACKET PRODUCTS

The main SCTGEN GUI Menu Panel is displayed, shown in Figure 3-65. The EDOS Packet Products option is selected and this scenario will display the Input Window Definition Menu Panel, shown in Figure 3-66.

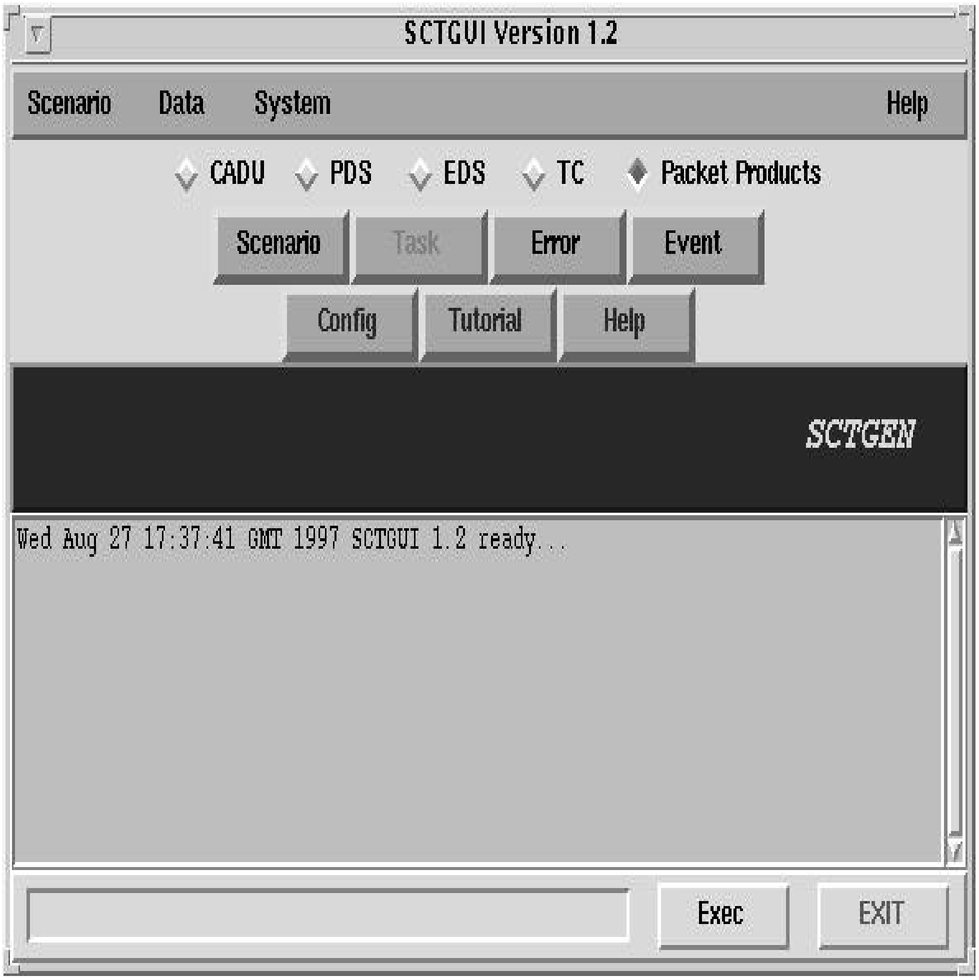

Figure 3-65. SCTGEN GUI Main Menu Panel

This menu panel will allow the user to select the creation of either an EDOS Data Unit (EDU) Stream or a Rate Buffered Product (RBP). Once the selection is made the appropriate window panel will be displayed. The following paragraphs will describe the sequence of steps to create EDOS Data Units and Rate-Buffered Product File.

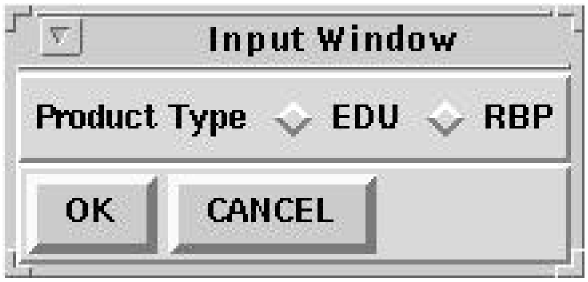

Figure 3-66. EDOS Packet Product Definition Input Window

3.6 EDOS DATA UNITS

When the EDUs option is selected the EDOS Input Window Definition Menu Panel, shown in Figure 3-67, will be displayed. This menu panel will allow the user to enter in the Spacecraft Identifier, and the VCID. Once this is entered, OK will take the user to the Scenario Definition Menu Panel displayed in Figure 3-68.

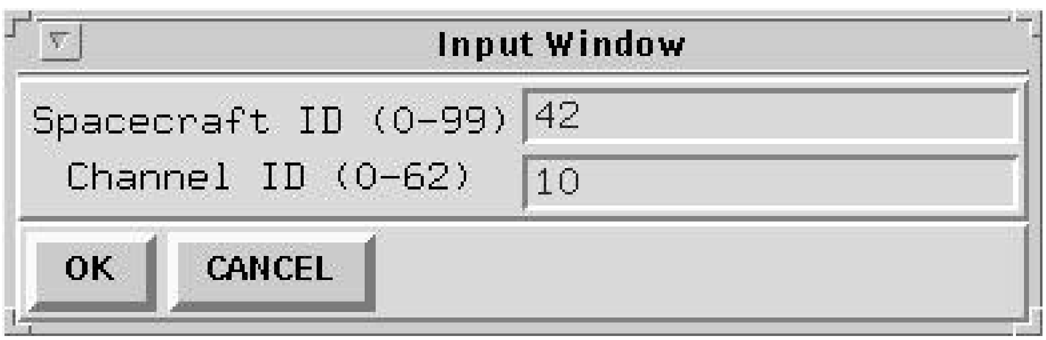

Figure 3-67. EDOS Data Units Input Window

The user can enter the APID, VCID and Packet Length to create the EDOS Data Units. If the user only wants a single APID stream, only one APID need be selected. Each APID entry needs to be registered by selecting the addAPID option.

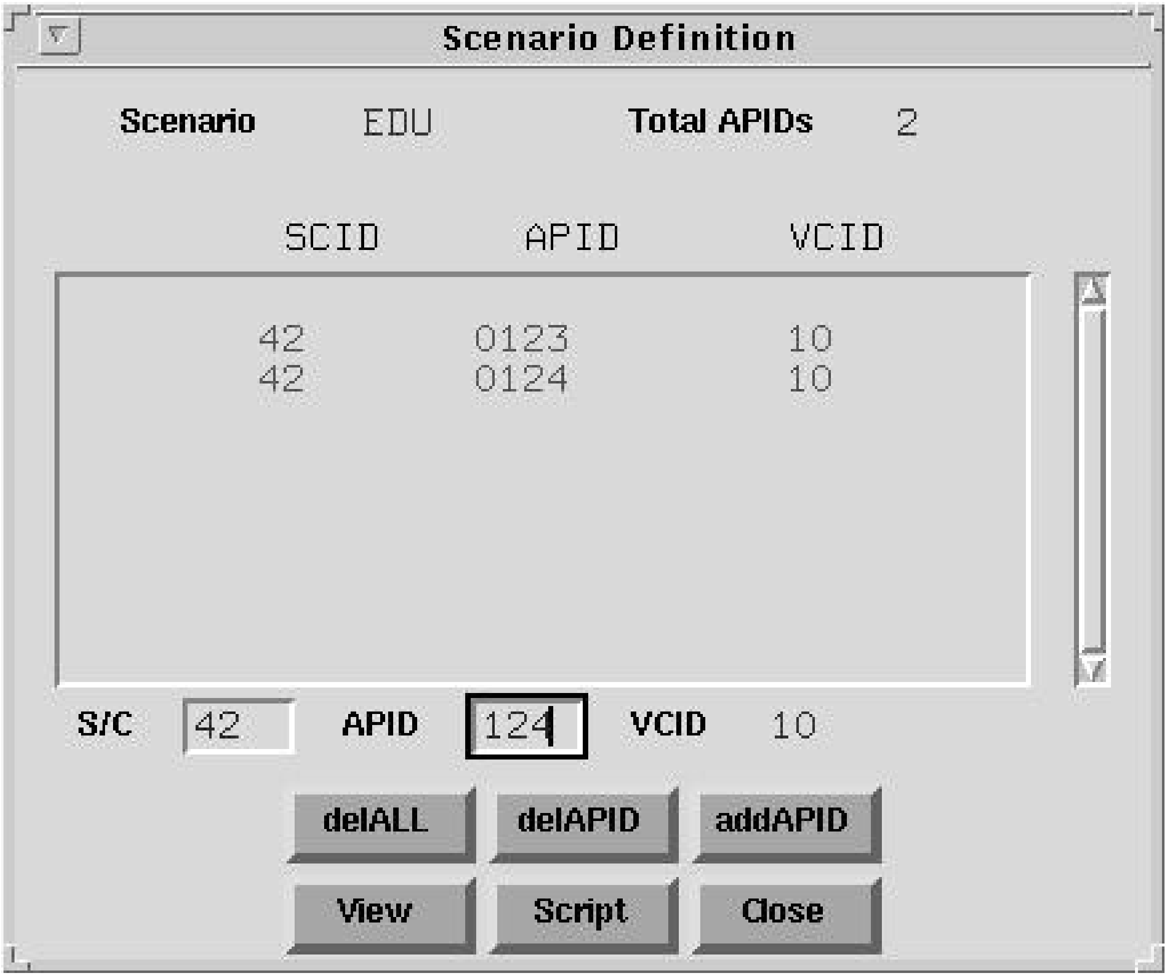

Figure 3-68. EDOS Data Units Scenario Definition Menu Panel

Once all the APIDs are entered, the view option will display the EDU Stream Menu Panel, shown in Figure 3-69.

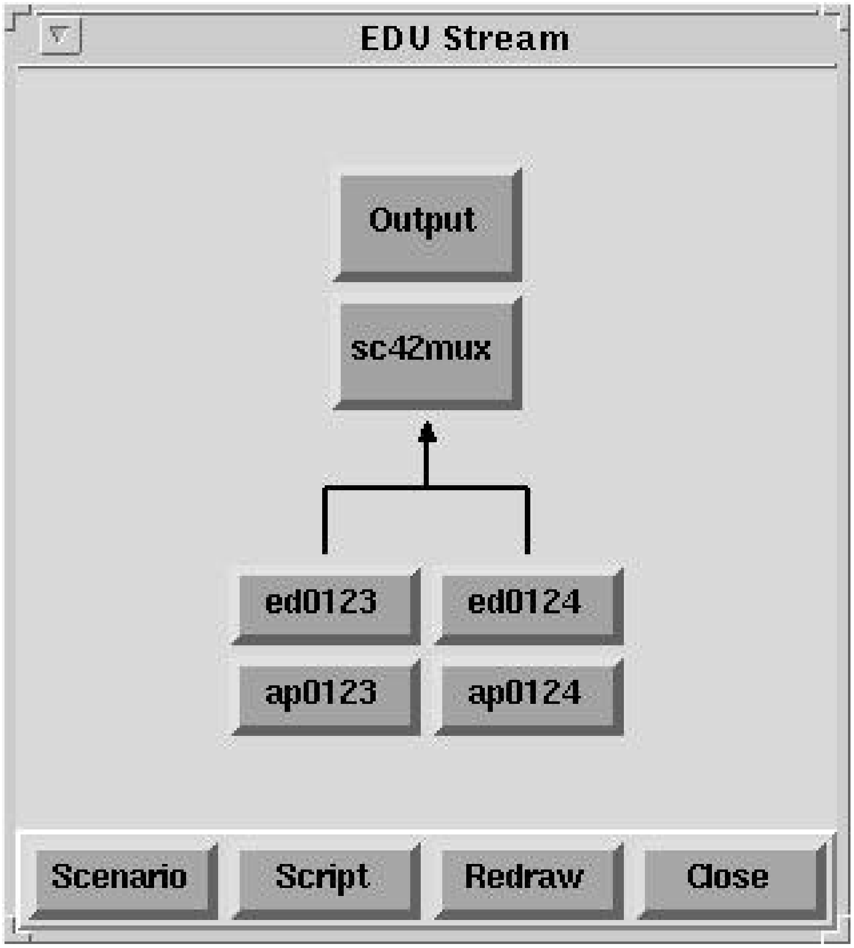

Figure 3-69. EDOS Data Units Stream Menu Panel

The first block to be defined in this menu panel is the Output block. When selected, this displays the Output Definition Menu Panel, shown in Figure 3-70. The first fields to be specified are the name of the stream file, the maximum number of units, which in this case is EDUs, and the option to have 'w' access on the target file. This menu panel also validates the instream that feeds this block. OK will register the user-inputs and CLOSE will take the user back to the EDU Stream menu panel.

The next block to be specified is the 'sc42MUX', which basically allows the user to specify the multiplexing strategy used to create the EDU output stream. If only one APID had been selected, there is no multiplexing required. However, the option to have more than one APID is available and thus the Packet Multiplexer Definition Menu Panel, shown in Figure 3-71, will be displayed. To illustrate the versatility of the tool, the multiplexed example has been displayed.

As described in the previous sections, the multiplexing is accomplished by one of two ways. The user can either select ranges for the each APID or select a repeating pattern.

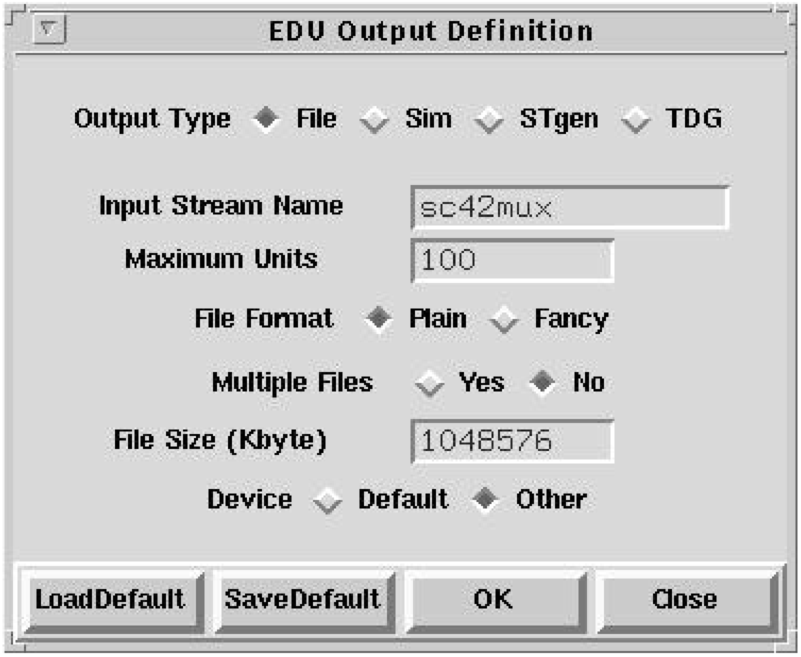

Figure 3-70. EDOS Data Units Output Definition Menu Panel

Once the pattern is selected and entered, the packet stream display will turn yellow for the defined units. Once the whole stream multiplexing strategy is complete, close will register the user-input and take the user back to the EDU Stream Menu Panel shown in Figure 3-69.

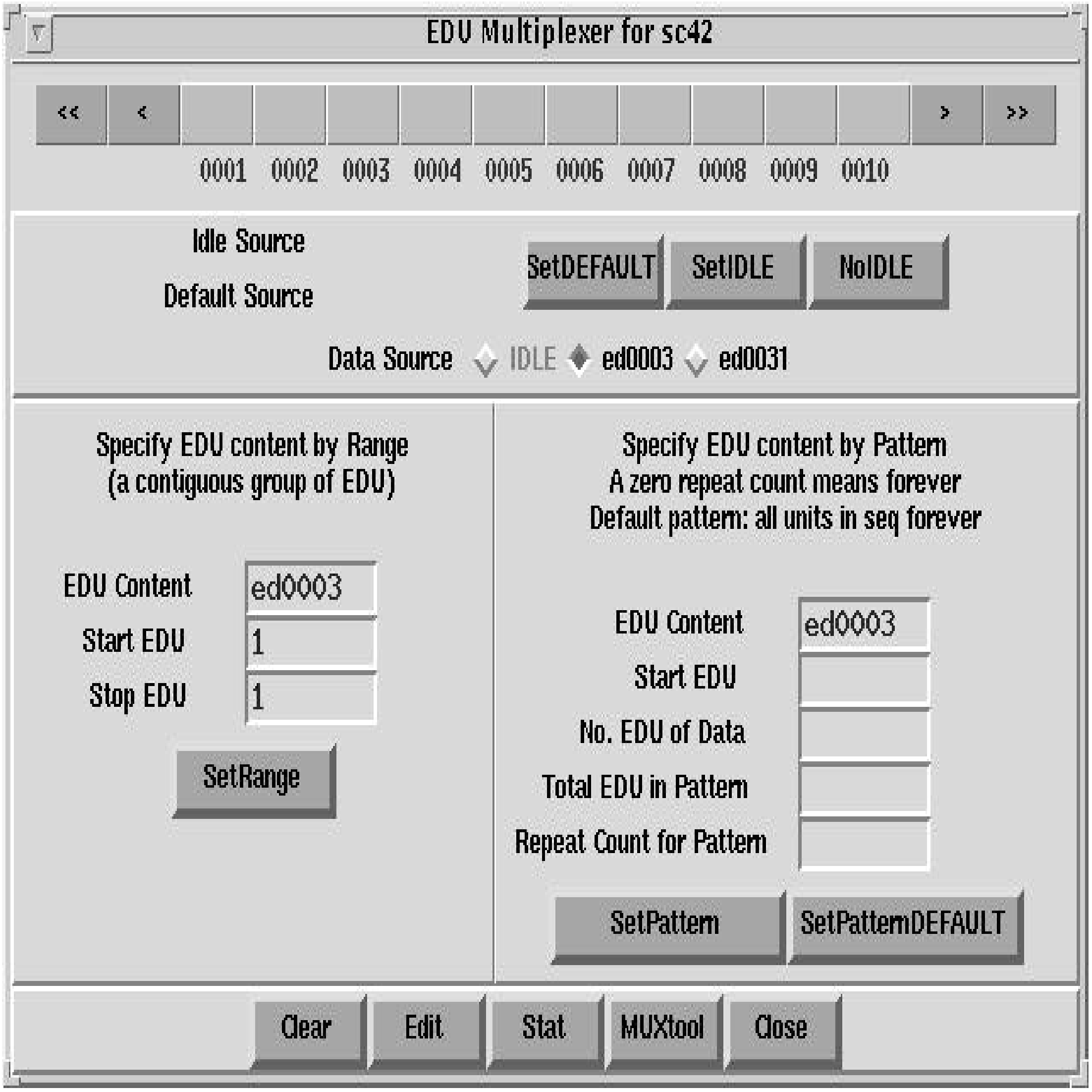

Figure 3-71. Packet (EDU) Multiplexer Menu Panel

Finally, the EDUs have to be defined. The EDU is essentially a packet with an EDOS Service Header. The 20-byte ESH for each APID is defined by selecting this block on the EDU Stream Menu Panel. This selection will display the EDU Stream Definition Menu Panel, shown in Figure 3-72.

The first few fields to be specified, namely 'version', 'vcid', 'spid', 'fill' and 'pb5' are the same as those for the PDS and EDS Output Definitions. There are two additional fields to be defined in this case, namely whether the EDU contains a packets from a playback stream or not, and whether the packet is a real-time downlink or a PB replay. To define the 'pb5' field in the header, the Set PB5 Values button is selected. If this is not selected, a default value will be entered in this field. When this option is selected the EDU PB5 Definition Menu Panel is displayed, shown in Figure 3-73.

In addition to these fields the user has to specify if the EDU stream came from a VCDU stream that had breaks, i.e., missing VCDUs to cause a break in the VCDU sequence count and hence drop packets in the stream. The user also has to specify if there were RS Uncorrectable CADUs and RS Correctable CADUs. In the case of the RS Uncorrectable CADUs, it is assumed that the user wants packets from these CADUs, and hence the ESH header for the packets extracted from CADUs tagged with the RS Uncorrectable event will have this field set in their respective ESH.

The Event Listing Menu Panel, shown in Figure 3-88 is displayed to enable the user to add and event. The 'AddEvent' option opens the Event Definition Menu Panel. As before all these events have to be defined and labeled and once again for each of these events the user must select 'addSpec' before exiting the Event Definition Menu Panel.

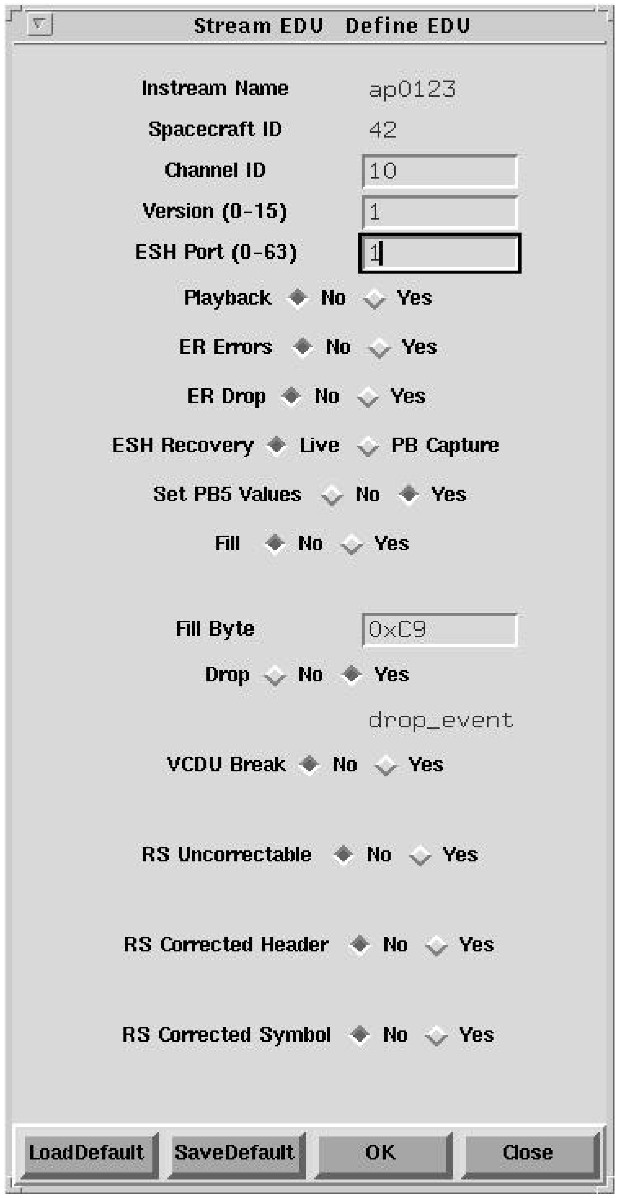

Figure 3-72. EDU Stream Definition Menu Panel

In the Event Listing Menu Panel, shown in Figure 3-74, the event must be selected before the window is closed to register the event name in the script.

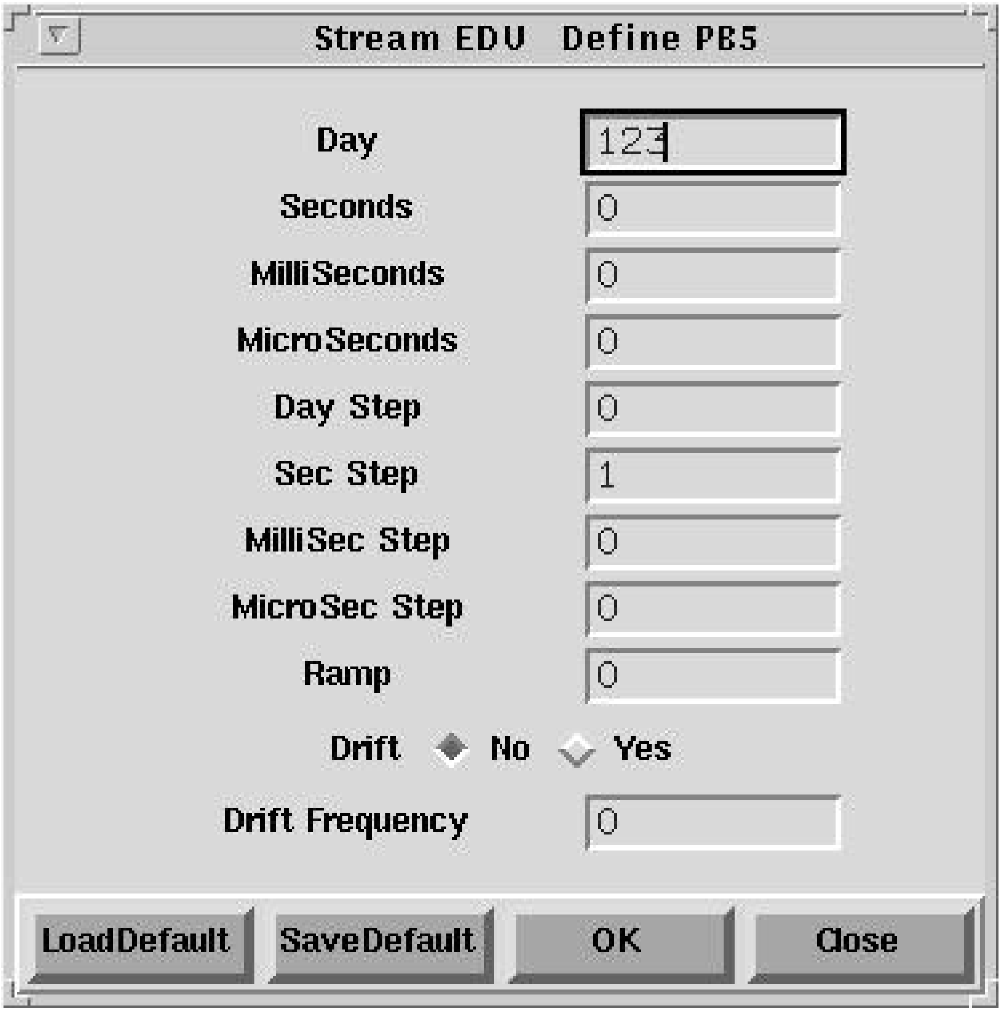

Figure 3-73. EDU PB5 Definition Menu Panel

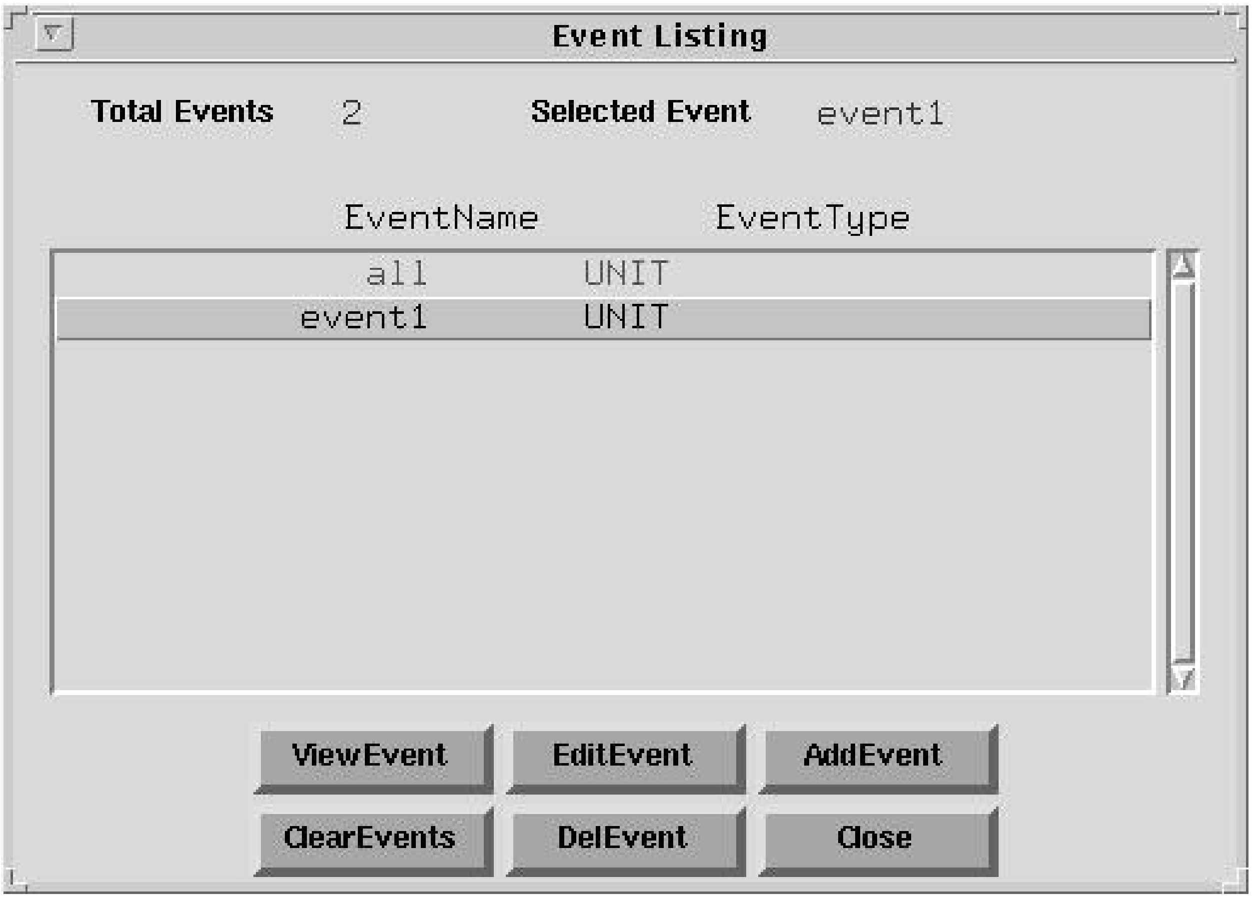

Figure 3-74. Event Listing Menu Panel

The selection of the 'apxxxx' button on the EDU Stream Menu Panel will display the Packet Definition Menu Panel in much the same way as they have been for all the scenarios described in this document. For completeness, the main Packet Definition Menu Panel is displayed in Figure 3-75, and the Error Listing and Definition is displayed in Figures 3-76 and 3-77.

Once the errors have been defined the Event Definition Menu Panel is displayed, as shown in Figure 3-78. As before after defining the event the 'AddSpec' option has to be selected before the OK button to enter and register the user input.

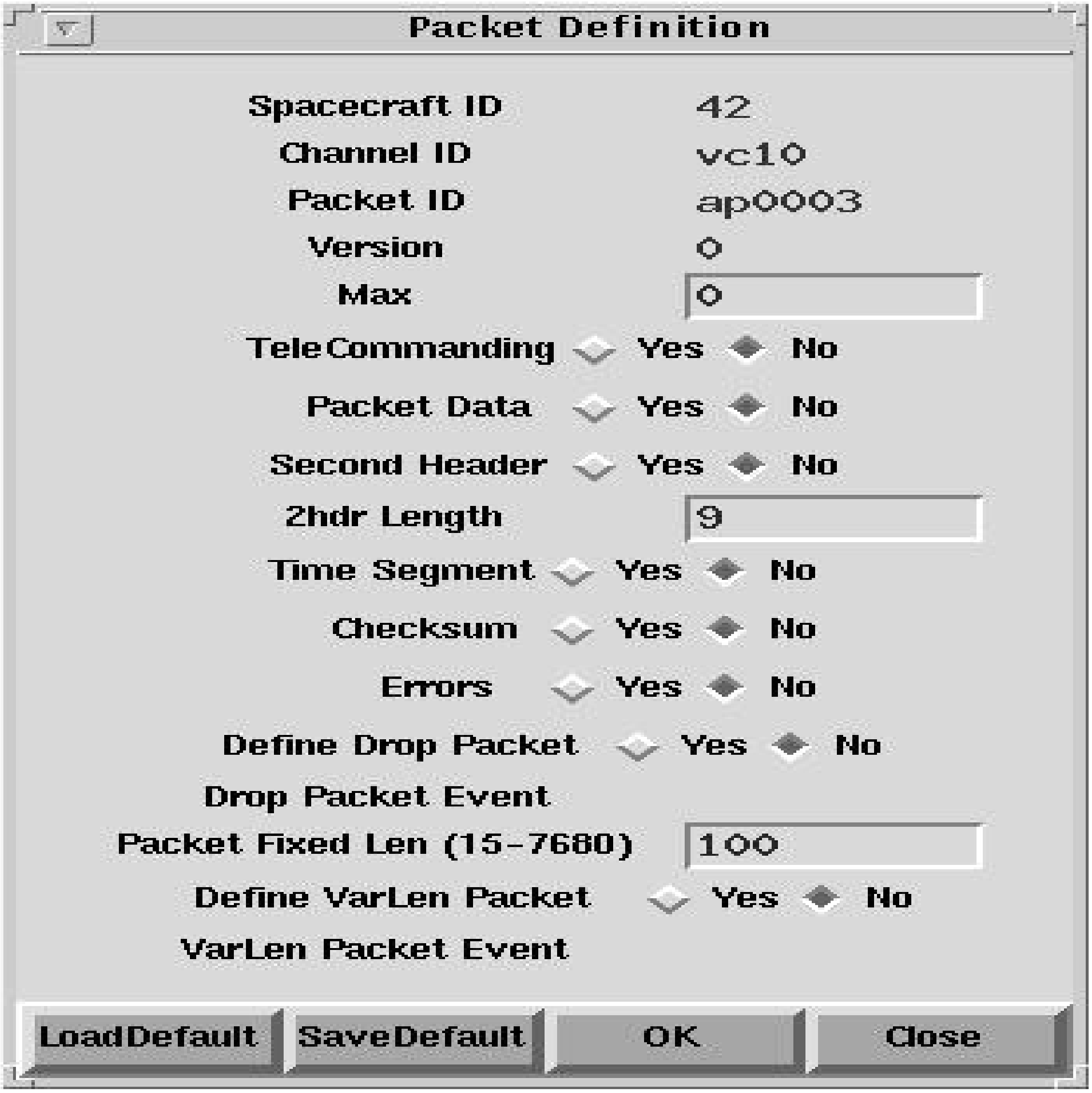

Figure 3-75. Packet Definition Menu Panel

OK and CLOSE will register the user-input and take the user back to the EDU Stream Menu Panel in Figure 3-69. The ESH for the next APID if any is specified in the same way. Once the ESH for all the specified APIDs are defined the user is in a position to define the Packet Stream for each APID.

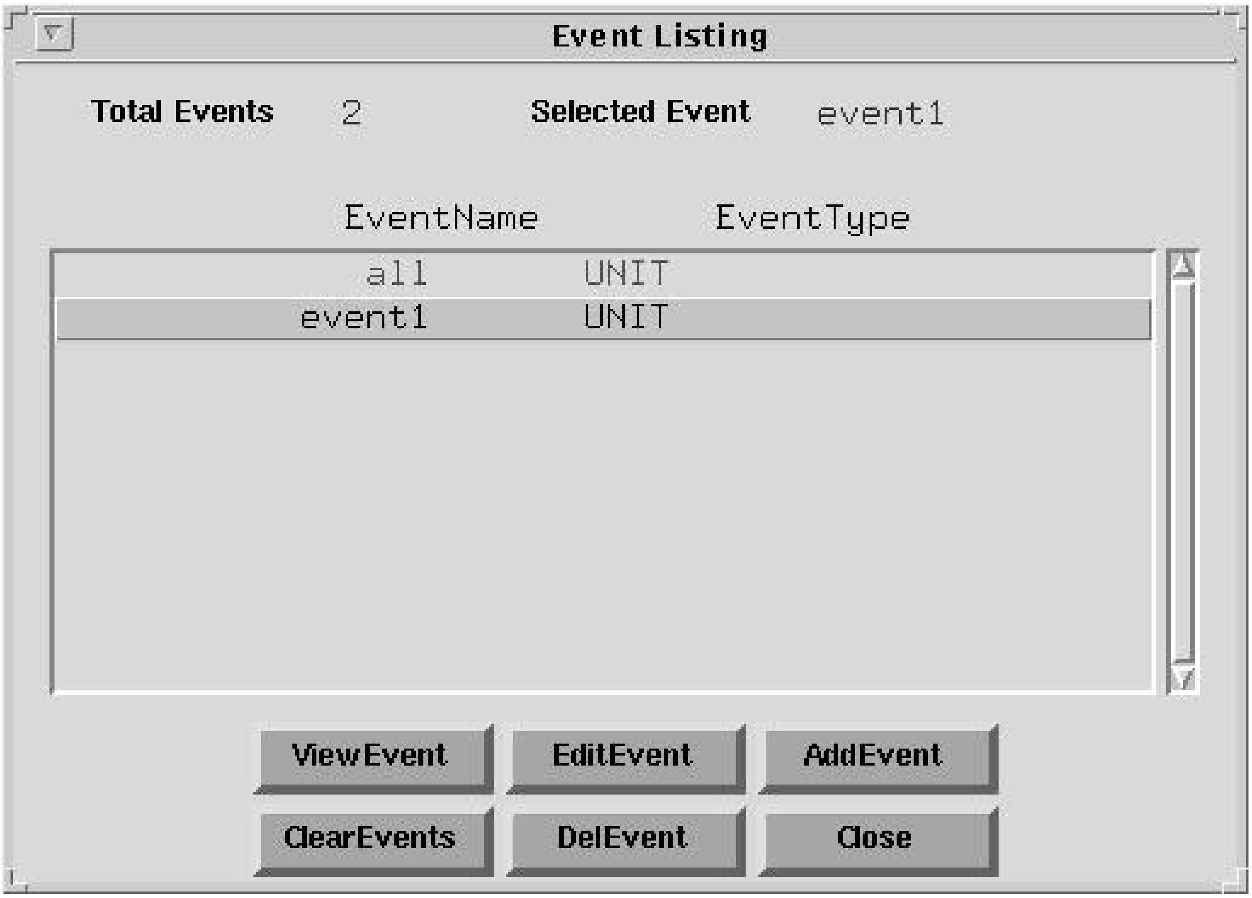

Figure 3-76. Error Listing Menu Panel

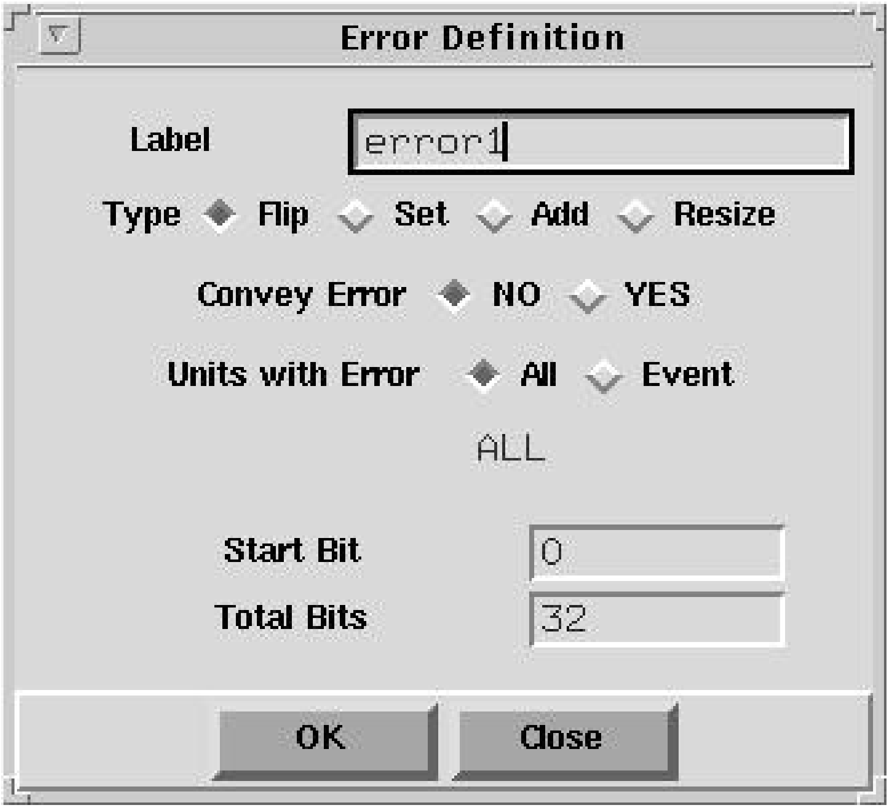

Figure 3-77. Error Definition Menu Panel

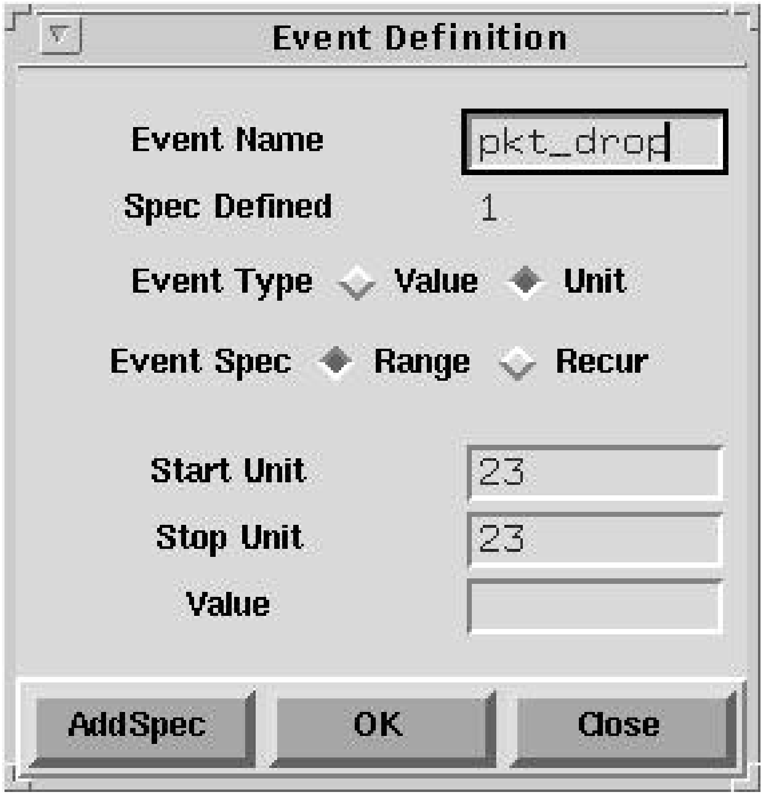

Figure 3-78. Event Definition Menu Panel

OK and CLOSE will register and take the user back to the menu panel displayed in Figure 3-66.

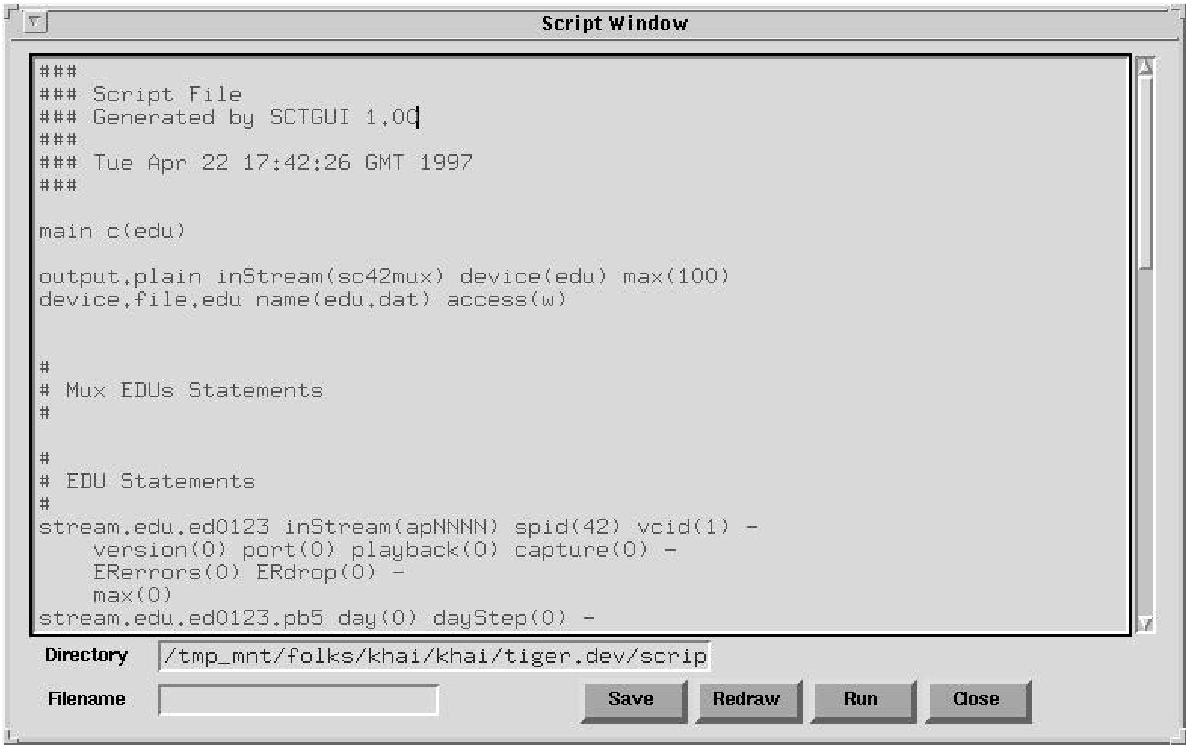

Figure 3-79a. EDOS Data Units Script Window Panel

After all the APIDs have been defined, the script button is selected to display the script that has been created. The display is depicted in Figure 3-79a. The user can briefly examine the script before creating the EDU Stream. At this point the user can also edit the displayed script to point the target to a null file, whereby the data is not actually created, but the expected results can be viewed as if the data has been created. When the 'run' option is selected the main SCTGEN Window show the event message that is displayed in Figure 3-79b.

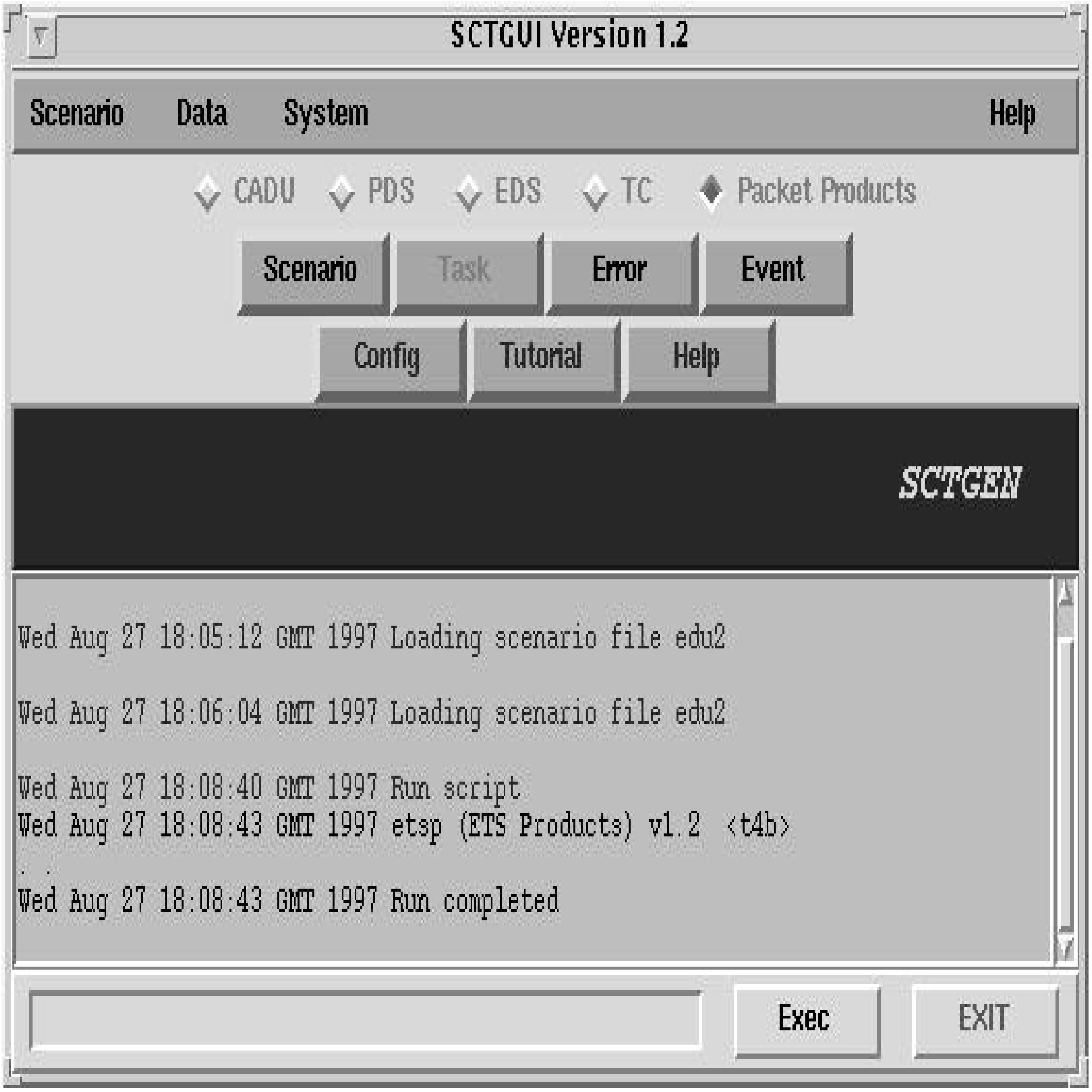

Figure 3-79b.   EDOS Data Units Run Window (Cont'd)

3.7 RATE-BUFFERED PRODUCT

When the RBP option is selected the RBP Input Window Definition Menu Panel, shown in Figure 3-80, will be displayed. The user can enter the Spacecraft ID, Channel or VCID, Application ID and Packet Length to create the EDOS Data Units that are in the RBP. In the RBP, only one APID has to be defined for each RBP File. These entries need to be registered by selecting the OK option which will take the user to the RBP Stream Menu Panel, shown in Figure 3-83. Each RBP file consists of a stream of EDUs from a single APID. This stream is saved as a RBP file with a unique RBP file-name.

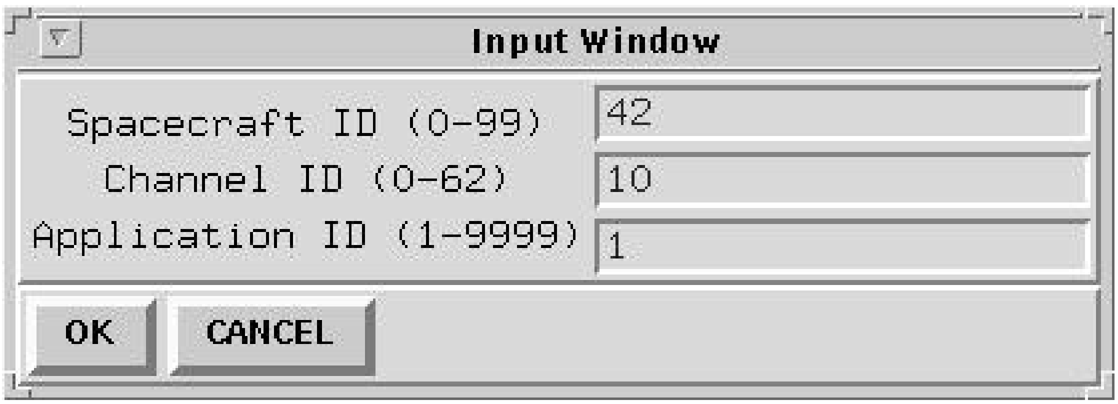

Figure 3-80. RBP Input Window

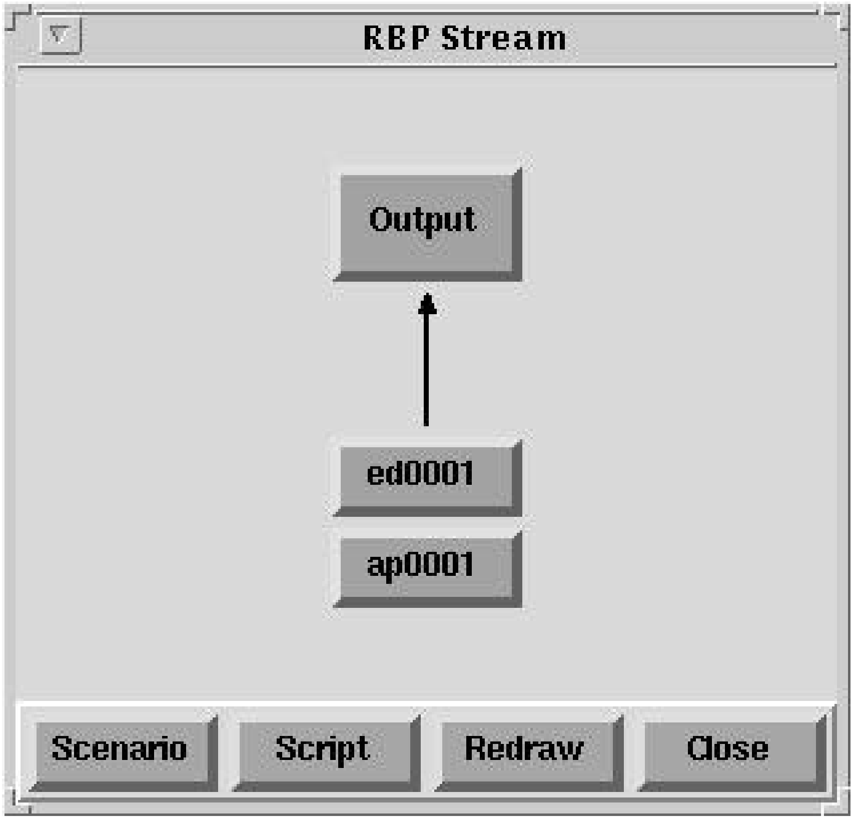

Figure 3-81. RBP Stream Menu Panel

The first block to be defined in this menu panel is the Ouptut block. When selected, this displays the Output Definition Menu Panel, shown in Figure 3-82. The first fields to be specified are the name of the stream file, the maximum number of units, which in this case is EDUs, and the option to have 'w' access on the target file. In addition, similar to the PDS, EDS, this menu panel allows the specification of the 'ground' station, the 'size' per file and the creation date. This menu panel also validates the instream that feeds this block, shown in Figure 3-83. OK will register the user-inputs and CLOSE will take the user back to the RBP Stream menu panel.

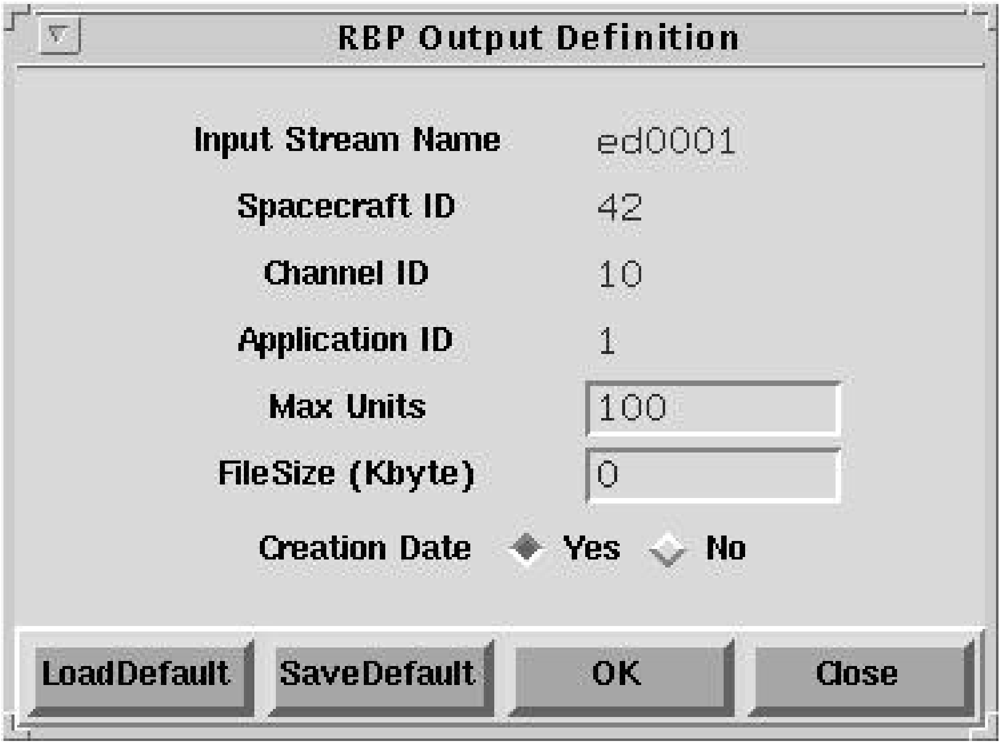

Figure 3-82. RBP Output Definition Menu Panel

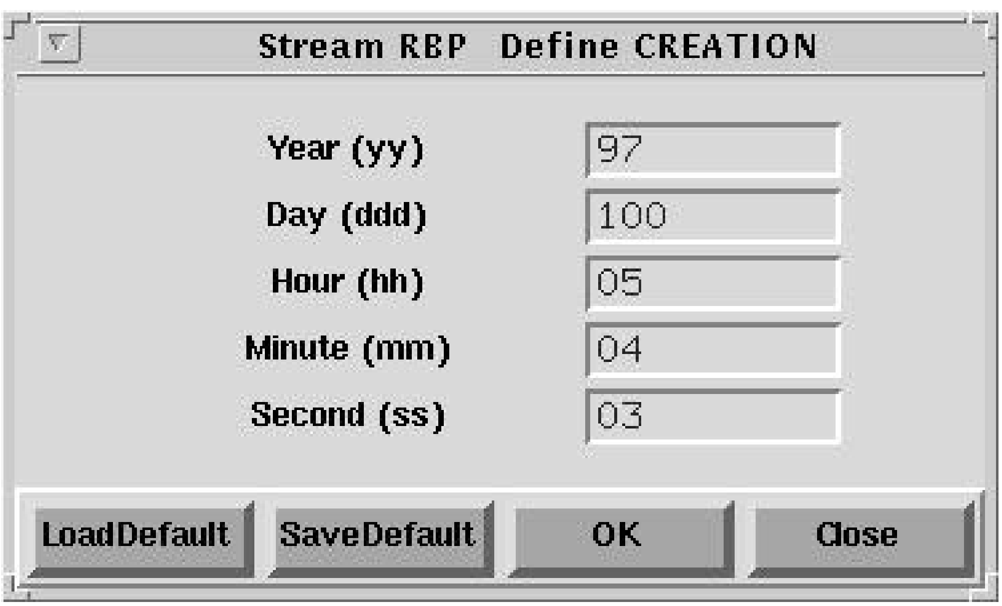

Figure 3-83. RBP Creation Definition Menu Panel

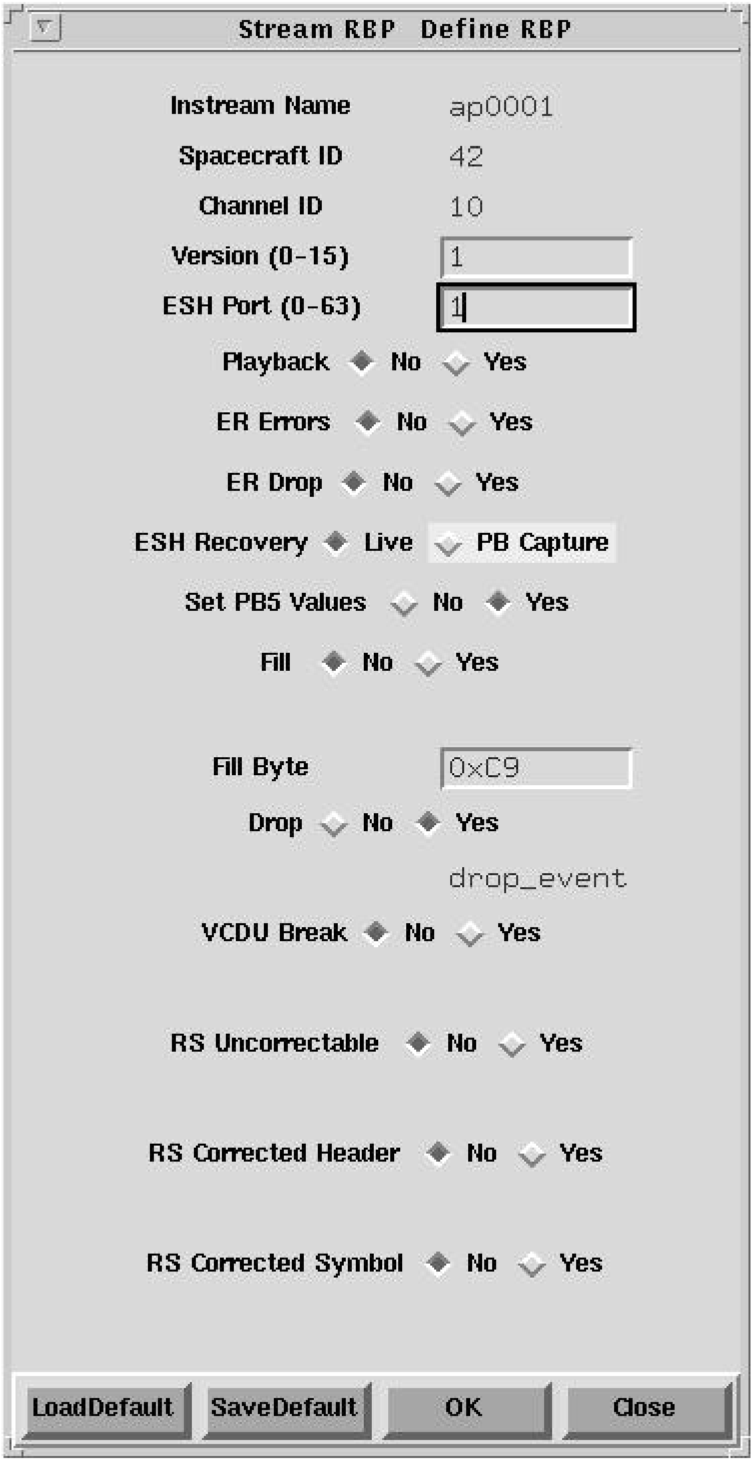

Figure 3-84. RBP Stream Definition Menu Panel

Finally, the EDUs within the RBP have to be defined. The EDU is essentially a packet with an EDOS Service Header. The 20-byte ESH for the APID is defined by selecting this block on the RBP Stream Menu Panel. This selection will display the RBP Stream Definition Menu Panel, shown in Figure 3-82.

The first few fields to be specified, namely 'version', 'vcid', 'spid', 'fill' and 'pb5' are the same as those for the PDS and EDS Output Definitions. The PB5 Value, if not set will carry a default value. When the option to set is selected, the PB5 Definition Menu Panel is displayed, as shown in Figure 3-85. OK will register the user input and CLOSE will take the user back to the Stream Definition Menu panel.

Figure 3-85. RBP Stream PB5 Definition Menu Panel

In addition to these fields the user has to specify if the EDU stream came from a VCDU stream that had breaks, i.e., missing VCDUs to cause a break in the VCDU sequence count and hence drop packets in the stream. The user also has to specify if there were RS Uncorrectable CADUs and RS Correctable CADUs. In the case of the RS Uncorrectable CADUs, it is assumed that the user wants packets from these CADUs, and hence the ESH header for the packets extracted from CADUs tagged with the RS Uncorrectable event will have this field set in their respective ESH.

When the user selects the option to include packets from ‘RSUncorrectable’, the Event Listing Menu Panel, as shown in Figure 3-86 will be displayed. The AddEvent option will open up the Event Definition Menu Panel, as shown in Figure 3-87. The ‘event name’ is entered and the type is selected. When unit is specified the range is all that is needed. As before all these events have to be defined and labeled and once again for each of these events the user must select ‘addSpec’ before exiting the Event Definition Menu Panel. Similarly, when the ‘RSCorrected Symbol’ is selected, the Event Listing Menu is once again displayed, and when ‘addEvent’ is selected, the Event Definition Window shown in Figure 3-86 is displayed. The ‘addSpec’ and OK buttons will register and enter the user inputs.

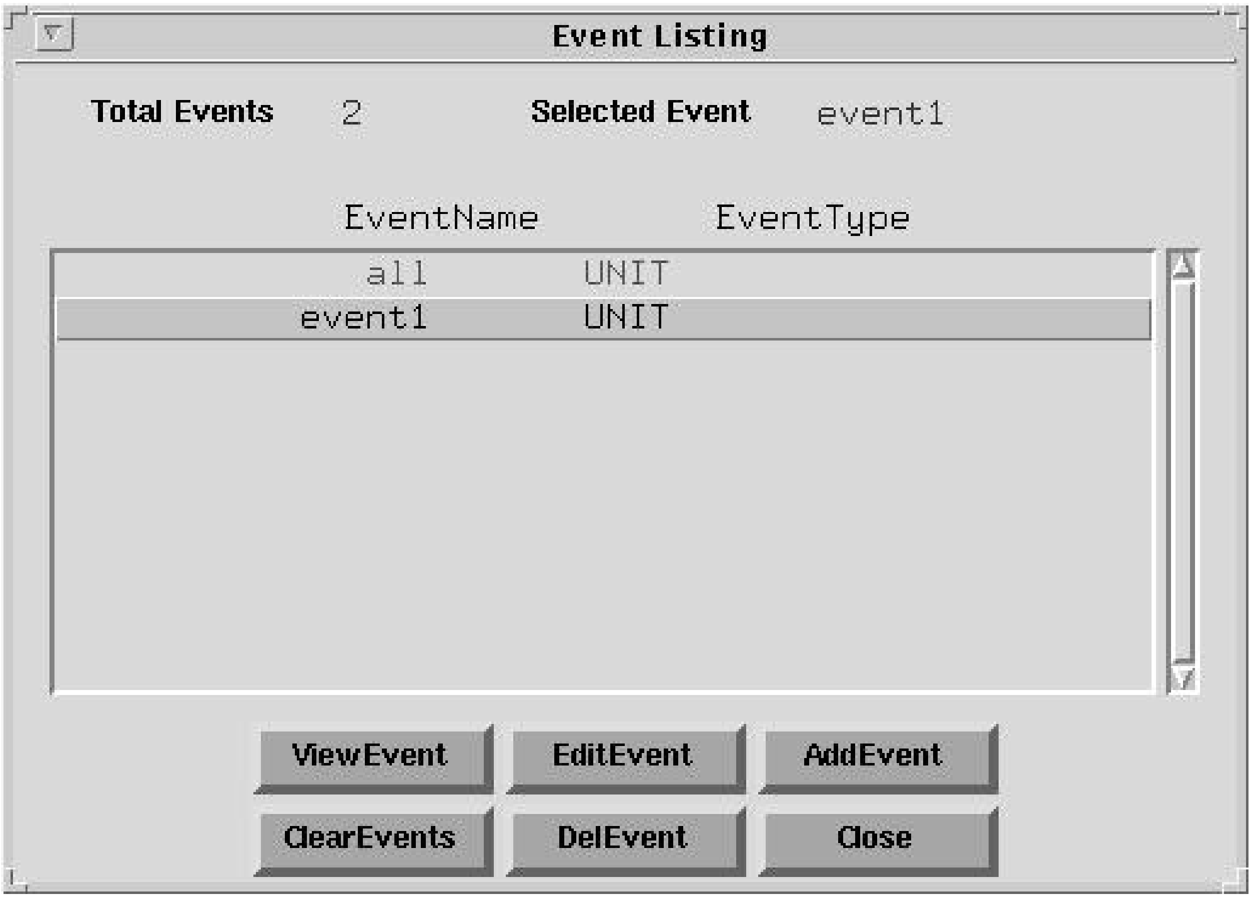

Figure 3-86. Event Listing Menu Panel

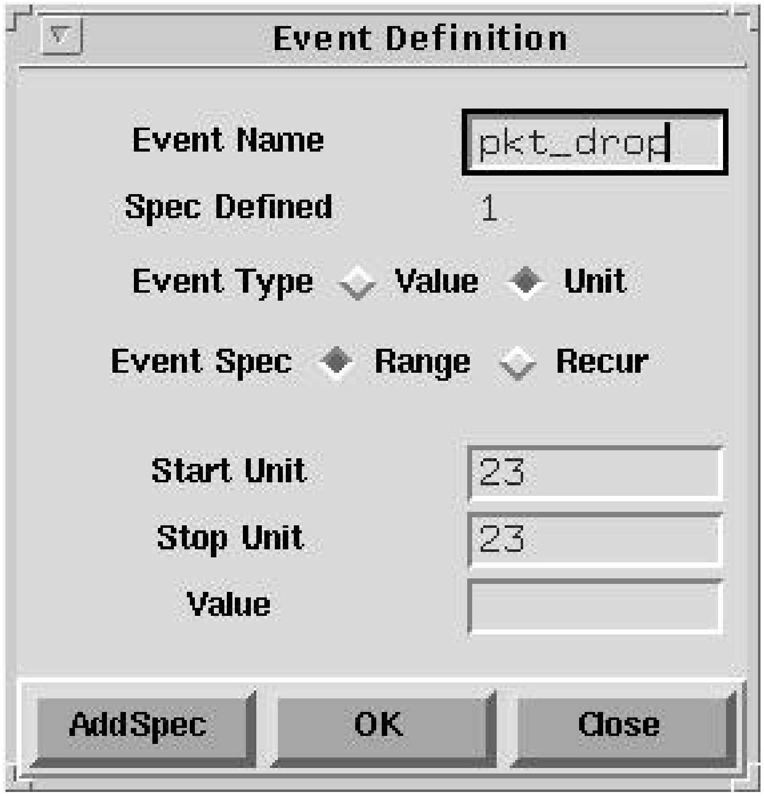

Figure 3-87. (RSUnc) Event Definition Menu Panel

In both Event Definition, in the Event Listing Menu Panel, shown in Figure 3-86, the event must be selected before the window is closed to register the event name in the script.

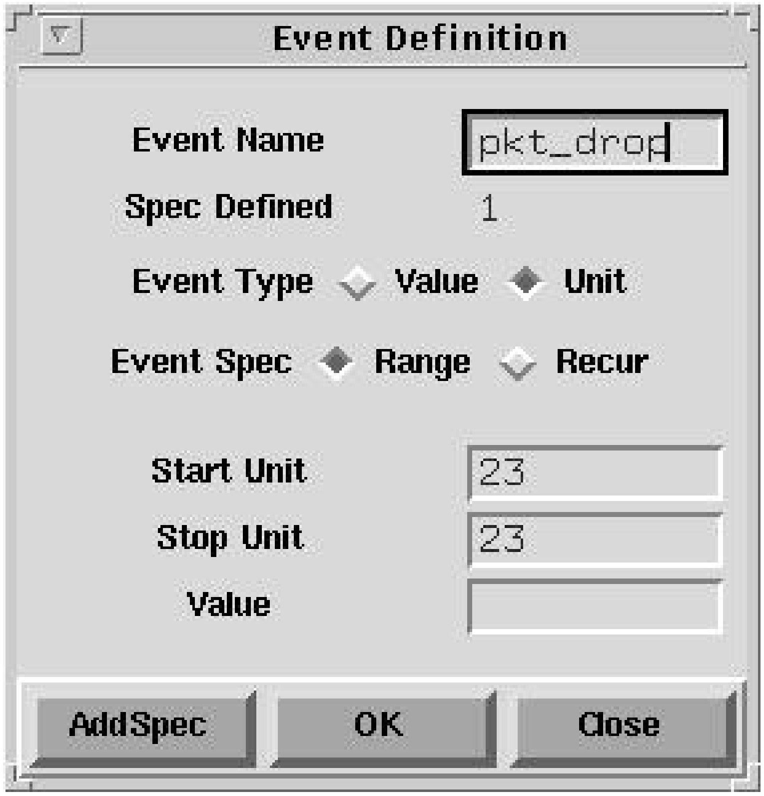

Figure 3-88. (RSCorr) Event Definition Menu Panel

OK and CLOSE will register the user-input and take the user back to the RBP Stream Menu Panel in Figure 3-81. Once the ESH for the specified APID is defined the user is in a position to define the Packet Stream for the APID. The selection of the 'apxxxx' button on the EDU Stream Menu Panel will display the Packet Definition Menu Panel in much the same way as they have been for all the scenarios described in this document. The main Packet Definition Menu Panel is displayed in Figure 3-89. Once the 'version' and 'max' fields are entered the user is now in a position to define the data region in the packet. Selecting the Packet Data option, the Data Definition Menu Panel, shown in Figure 3-90, is displayed. The pattern is selected in much the same way as it has been done for the other scenarios described in the previous sections of this document.

Once the Data region is defined the OK and CLOSE will take the user back to the Packet Definition Menu Panel. The user can now specify the Secondary Header in the same manner that has been done for the other scenarios, by selecting the option entering the field values. After verifying the packet length field, the user is now in a position to insert errors and drop packets from the RBP.

When the Error insertion option is asserted as 'yes', the Error Listing Menu Panel, shown in Figure 3-91, is displayed. The 'addError' option, will display the Error Definition Menu Panel, shown in Figure 3-92. The user can either choose to have all the packets as having the inserted error or select an event as to which packets should have it.

If the 'event' option is selected in the Error Definition, the Event Listing Menu Panel is displayed, Figure 3-93. The 'addEvent' option, opens up the Event Definition Menu Panel, Figure 3-94, and the same sequence of steps as described in the previous paragraphs, are followed to label the event, select the range options and register the user inputs. After the event is defined and the listing

selected to register the event, the user will select the CLOSE button to get back to the Error Definition Menu Panel. Here the user will select the type of error, i.e., flip or set etc.,. The position in the packet and the number of bits are entered and the OK and CLOSE will take the user back to the Packet Definition Menu Panel in Figure 3-89.

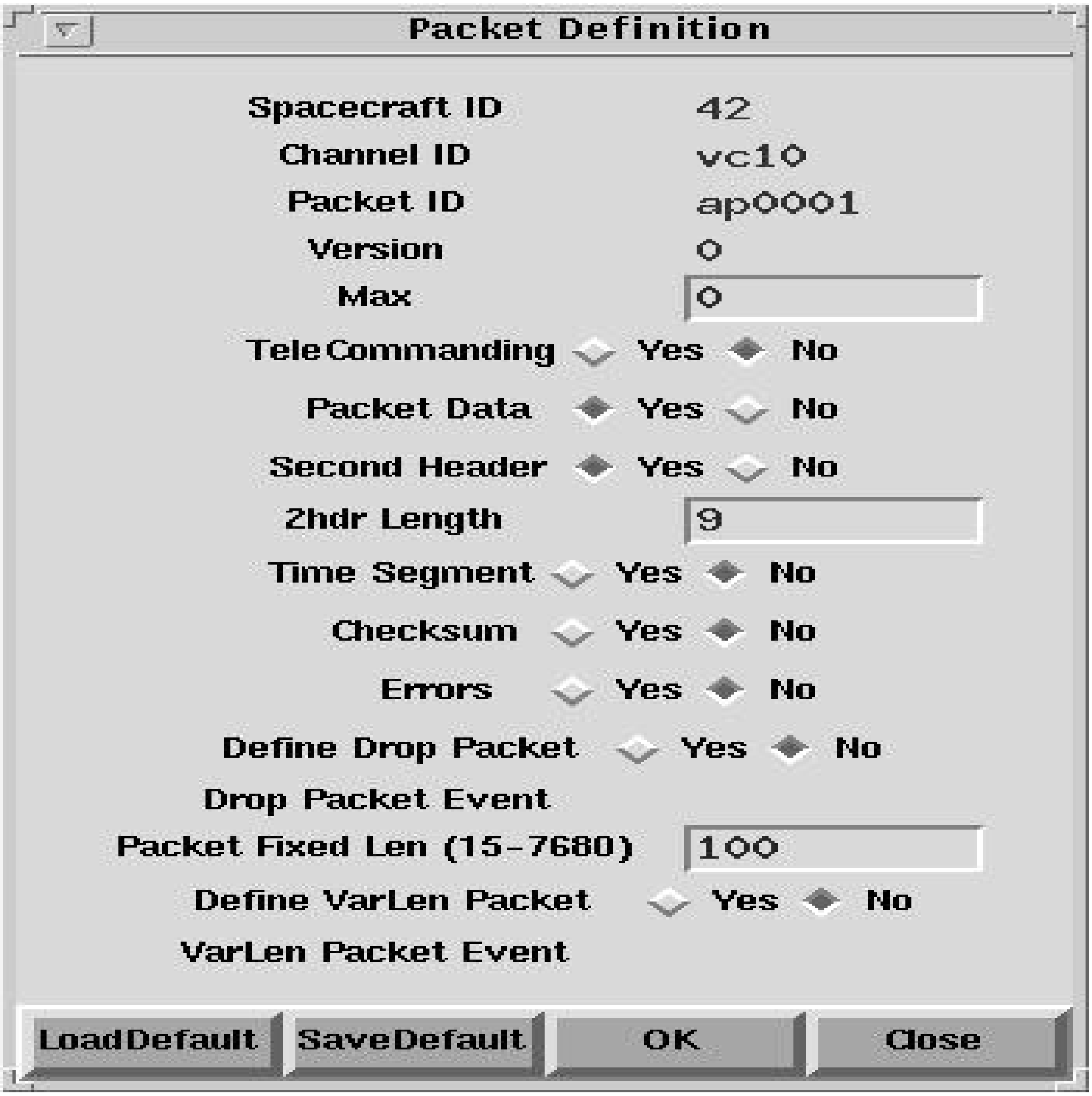

Figure 3-89. Packet Definition Menu Panel

OK and CLOSE will register and take the user back to the menu panel displayed in Figure 3-81. After the APID has been defined, the script button is selected to display the script that has been created. The display is depicted in Figure 3-95. The user can briefly examine the script before creating the RBP.

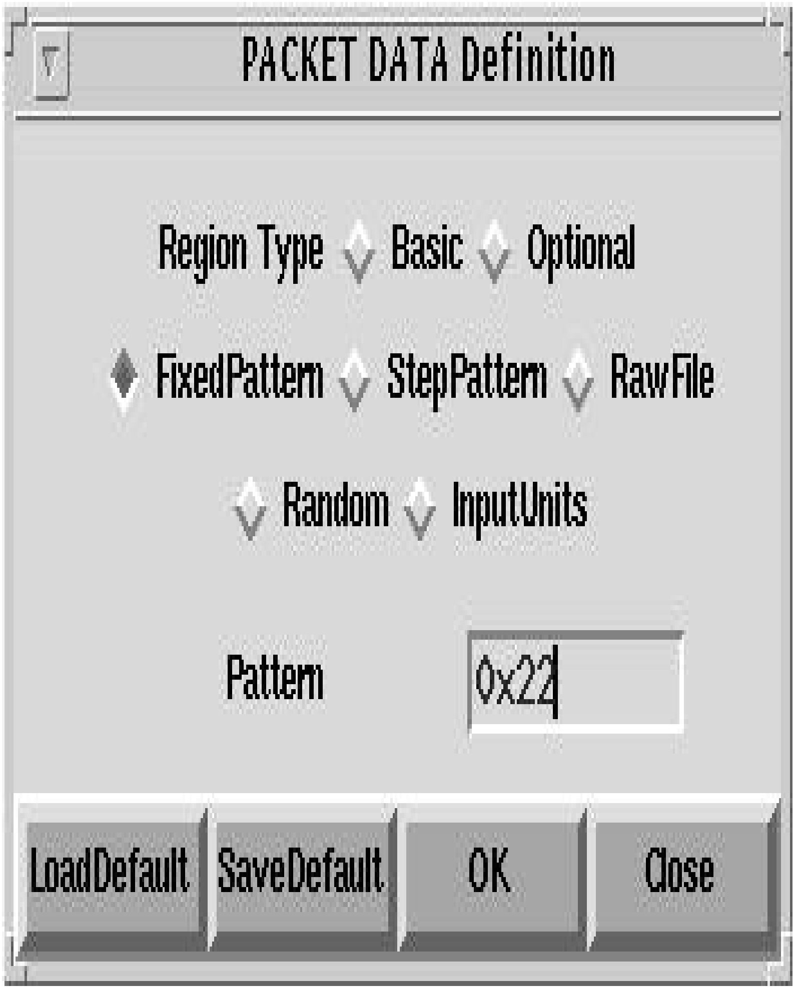

Figure 3-90. Packet Data Definition Menu Panel

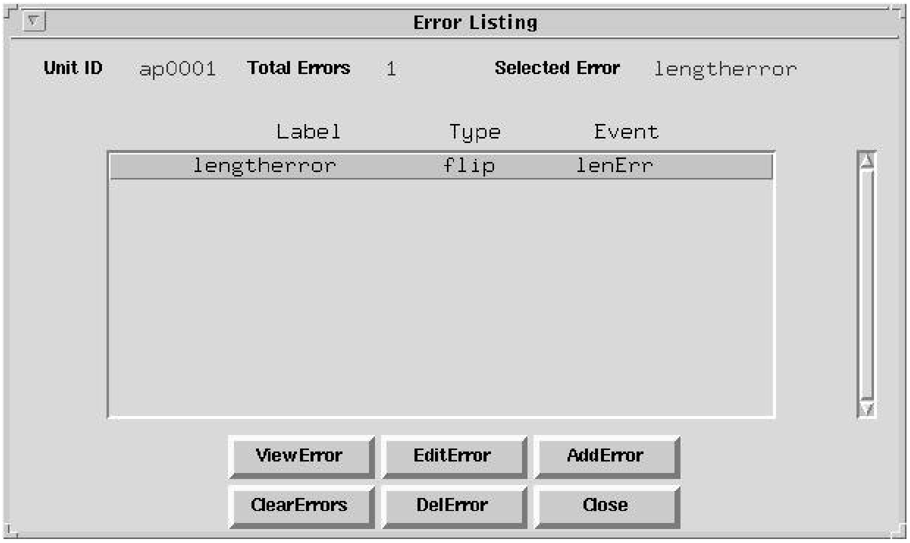

Figure 3-91. (Error) Listing Menu Panel

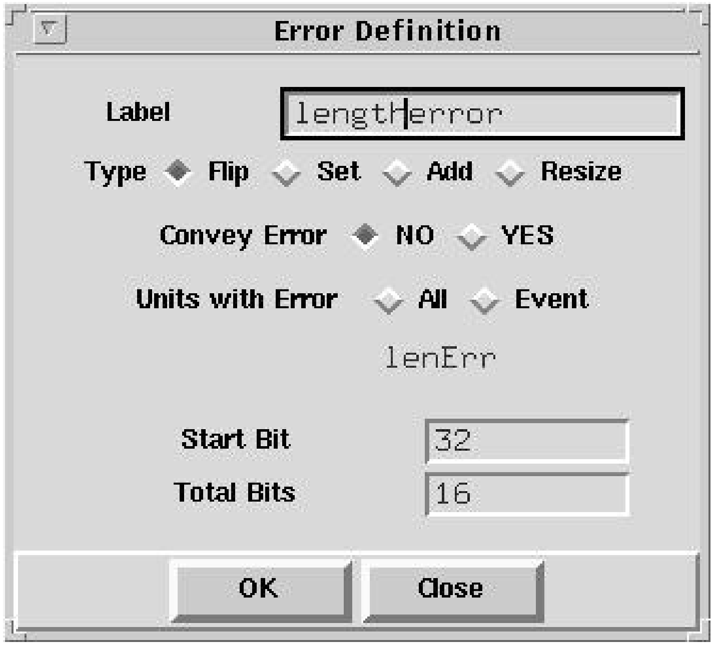

Figure 3-92. (Error) Definition Menu Panel

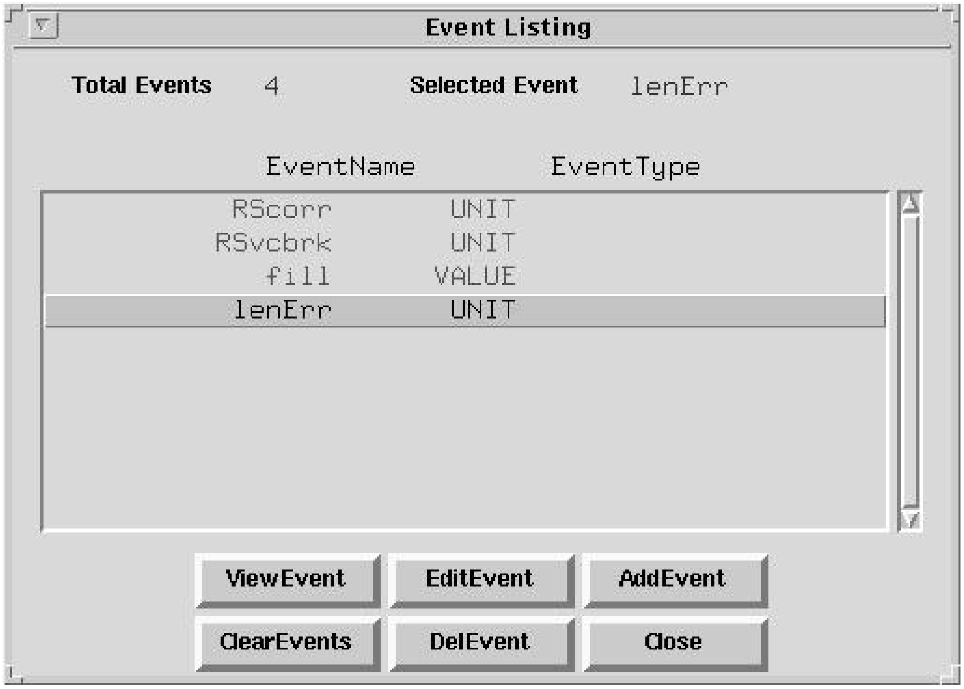

Figure 3-93. (Error) Event Listing Menu Panel

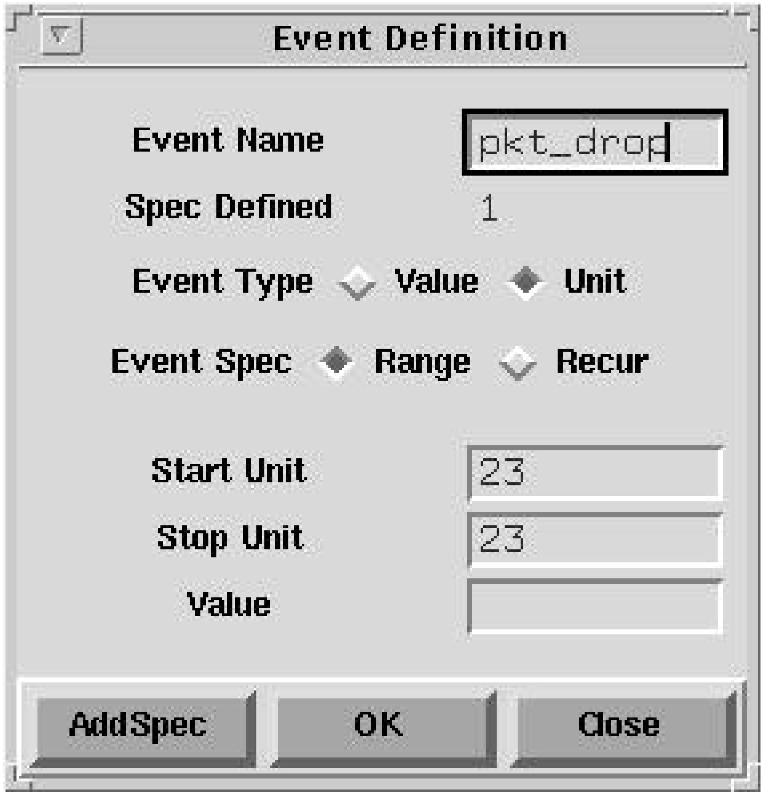

Figure 3-94. (Error) Event Definition Menu Panel

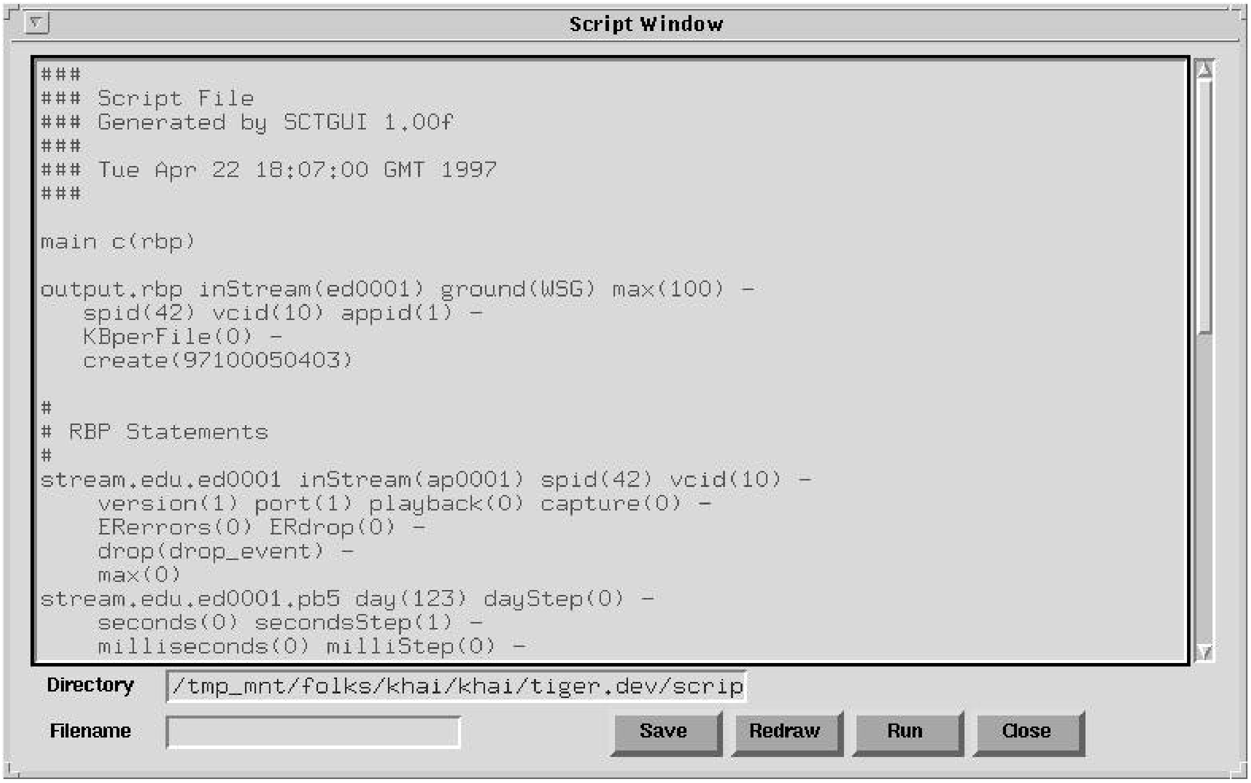

Figure 3-95. RBP Script Window Panel

At this point the user can also edit the displayed script to point the target to a null file, whereby the data is not actually created, but the expected results can be viewed as if the data has been created. When the 'run' option is selected the main SCTGEN Window show the event message that is displayed in Figure 3-96.

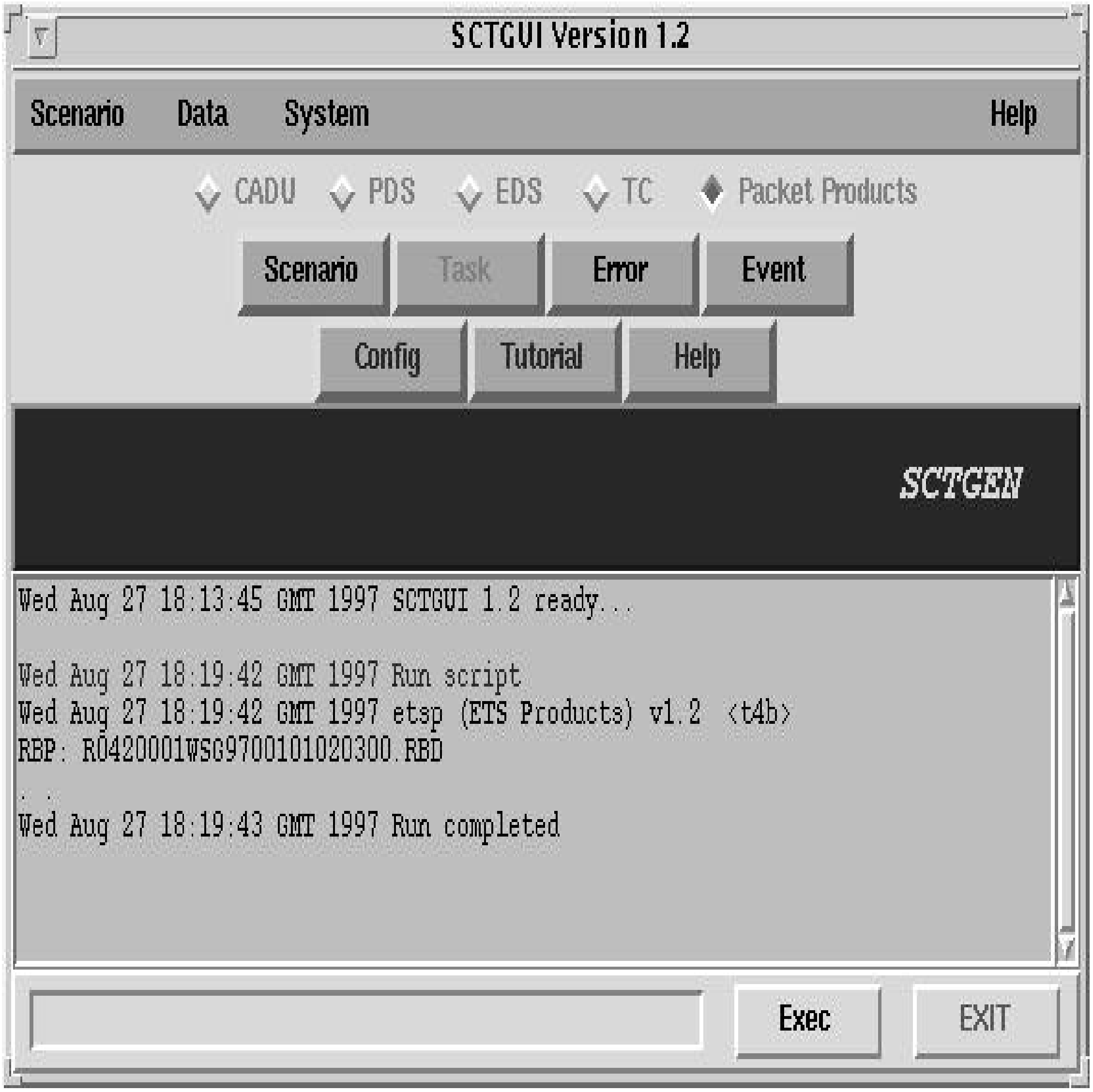

Figure 3-96. RBP Run Window

3.8 SCTGEN TUTORIAL

In Figure 3-97, when the Tutorial button is selected the SCTGEN Tutorial Menu Panel shown in Figure 3-98 is displayed.

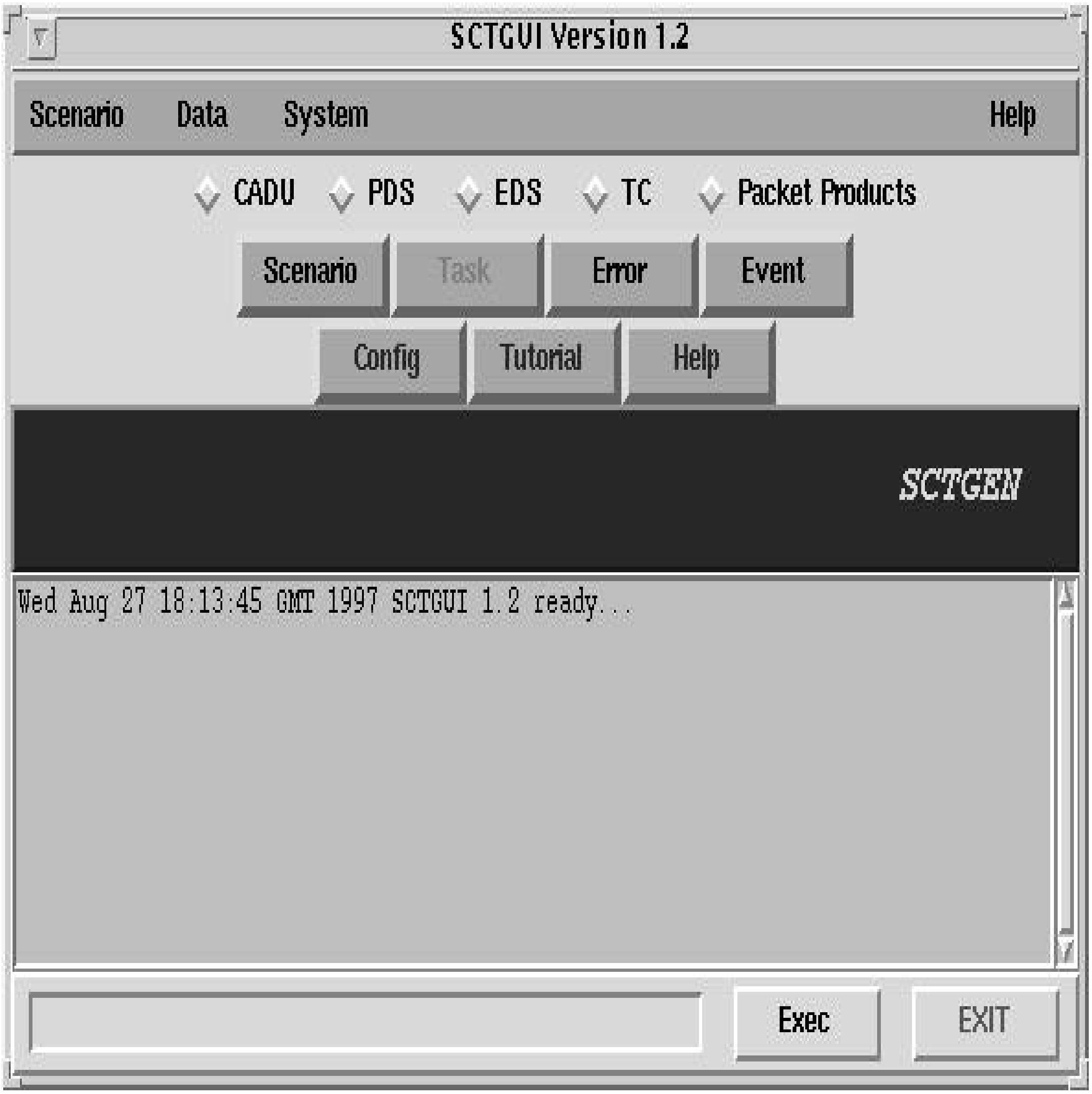

Figure 3-97. SCTGEN Main Menu Panel

The SCTGEN Tutorial screen as shown in Figure 3-97 allows the users to familiarize themselves with the different menu pages in the GUI. At present it is available only for the CADU script generation, and will be updated to include the other data scenarios.

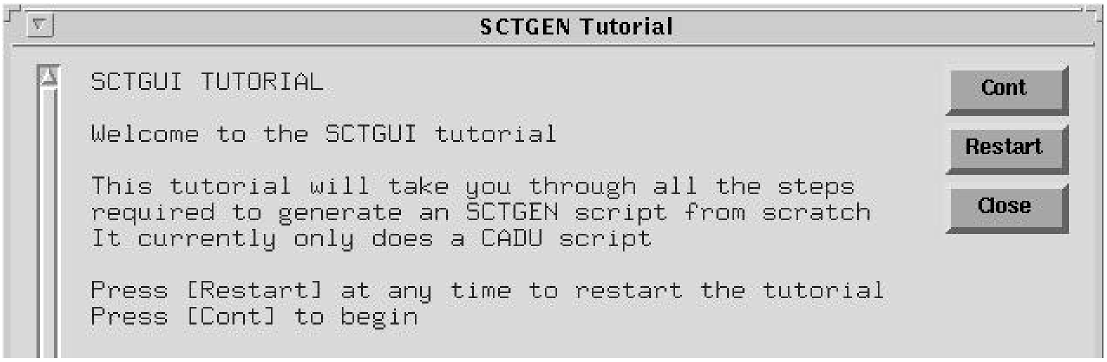

Figure 3-98. SCTGEN Tutorial Screen

3.9 SCTGEN HELP

The SCTGEN Help option allows the user to access any of the reference documents that are on the system. The path to access the help files as with all the other files in the system is configured using the Configuration Menu Panel which is displayed from the main Menu Panel.

In figure 3-97, when the Config button is selected, the Configuration Menu Panel, shown in Figure 3-99 is displayed.

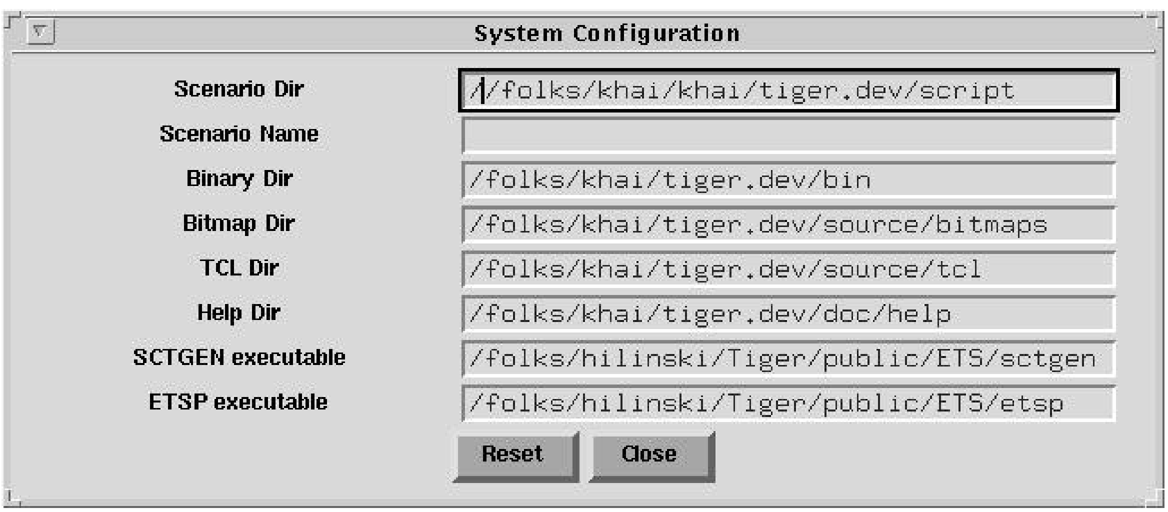

Figure 3-99. SCTGEN Configuration Screen

Once the path has been correctly specified, the selection of the Help button on the Main Menu Panel, shown in Figure 3-97, will display the Help Screen shown in Figure 3-100.

Selecting a topic from the listing on the screen will open a text file containing specific information on the topic. For example, opening the 'cadu' help topic, the user will be able to access the CADU refernce document which will display all the arguments that may be defined for a CADU stream.

It should be noted that GUI has been developed to adhere to the EOS-AM1 requirements for data generation and thus all the arguments cannot be specified using the GUI. However, when the script is veiwed using the script window, any argument may be typed in as long as it conforms to the SCTGEN specified syntax. Figure 3-101 shows the display when the 'cadu' help button has been selected.

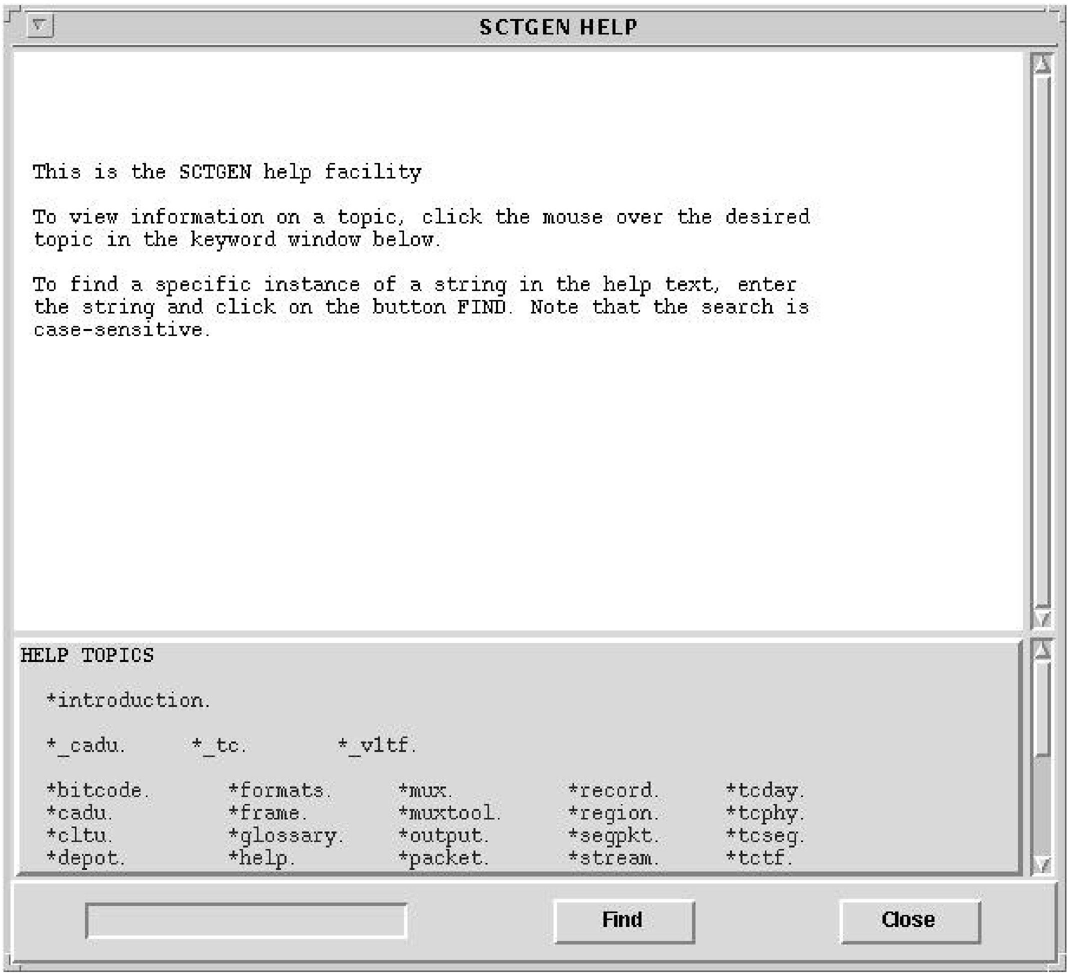

Figure 3-100. SCTGEN Help Screen

```
[cadu.doc]

CADU STREAM

This stream makes CADUs (frames). It may only be used in the CADU scenario.
See STREAM.DOC for general stream information.

The CADU stream can also read CADUs from a file instead of making them. You
may also use the depot stream to read CADUs from a file. See DEPOT.DOC.

The stream type must be "cadu." You may choose any name for <name> provided it
is different from any other stream name. The usual practice is to use the
virtual channel id in the stream name.

stream.cadu.<name> service(c) length(n) vcid(n) spid(n) sync(1or0:1) -
                   frameSync(0xnnnnnnnn:0x1acffc1d) syncBytes(n:4) max(n:0) -
                   replay(ename:0) hdrErrControl(1or0:0) fill(0xnn:0xc9) -
                   insertZone(n:0) ocf(1or0:0) insertPDUheader(1or0:1) -
                   PDUheaderBitLength(n) PNencode(1or0:0) RSencode(1or0:0) -
                   RSinterleave(n:4) RSdual(1or0:1) RScodeLength(n:255) -
                   RSempty(1or0:0) crc(1or0) CRCstart(0xnnnn:0xffff) -
                   CRCempty(1or0:0) CRCincludeSync(1or0:0) invert(c:N) -
                   invertEvent(ename) ERerrors(1or0:1) ERgaps(1or0:1) -
                   ERdrop(1or0:1) device(dname) fileType(c) stepSize(n:1) -

HELP TOPICS

  *introduction.

  *_cadu.     *_tc.       *_vltf.

  *bitcode.      *formats.     *mux.         *record.      *tcday.
  *cadu.         *frame.       *muxtool.     *region.      *tcphy.
  *cltu.         *glossary.    *output.      *segpkt.      *tcseg.
  *depot.        *help.        *packet.      *stream.      *tctf.

                              Find                    Close
```

Figure 3-101. SCTGEN CADU Stream Help Screen

3.10 OPERATIONAL TEST DATA GENERATION SCENARIO

Once the user is satisified that the expected results match the devised scenario for testing, the user will then generate the test data. This is done by targeting a file name and location where this data is to be stored, or a socket name if the storage device is the TRS. The maximum test data size is storage limited to 25 gigabytes. The format of the test data is selected by the user to be any one of the formats described in section 2, i.e., Plain file format, SIM format or any other types of SCTGEN formats available. The type of data, be it Command Blocks or CADUs, will be designated as the highest layer of data encapsulation.

In the instance that the test data is CADUs with Version 1 packets, the sequence of test data generation is as follows:

The user will first select the total number of frames to be generated, for example 100,000. The user will then assign a specific number of APIDs and VCIDs to these frames, specifiying number of packets per APID per VCID.

The next step will be to devise the frame multiplexing design to generate the test data, i.e., VCID 1 (10 CADUs) to be followed by VCID 2 and 3 in an alternating pattern, and so on. Within the VCID stream, the user will devise the multiplexing strategy based on packet rates or user-selected frequencies. Once this is done the user will either generate the pseudo-test data file and view the data to selectively insert errors or insert errors before the first generation cycle and generate the errored test data.

In the former option, once the test data are generated, the user will then devise the frame level modifications to simulate frame level errors, and packet level errors to simulate packet errors. Once all the frame and packet level modifications are entered, the user will re-generate the test data. Example scripts and the data generated using these scripts is appended at the end of this document.

SECTION 4
USER DETAILED REFERENCE DOCUMENTATION

These following paragraphs contain detailed information on what fields have to be filled in the script and what values are applicable. Each sub-section id referred to as a document within the paragraph.

4.1 READ-ME

This document set is the reference for Tiger script statements. If you are unfamiliar with Tiger, then see GLOSSARY.DOC for a definition of terms. For general information on script language syntax, see SYNTAX.DOC.

To create a new script, you can either use an old one as a template or you can use these documents to paste together a script. If you are using a windowing system such as X-windows, you can open a document in a window and copy and paste statements to your script. I have conveniently placed statement templates at the top of each file to make this task easier. Note that these documents refer to each other with names in caps, but the actual documents may be in lowercase or may use slightly different naming conventions.

To begin, open either _CADU.DOC, _V1TF.DOC, or _TC.DOC depending on the type of scenario you are trying to create. If you are using a special application, then begin by opening the document for that application. If you are creating a script that is independent of scenario type, such as making generic frames or packets, then use _CADU.DOC. The scenario type document will tell you which components are available and which documents you should open to get them.

Regardless of the scenario type, most applications such as tccsds use a general model for script layout. You will need a main statement, an output statement, and one or more stream statements. The main statement is described in MAIN.DOC. Some applications do not require a main statement, but it does not hurt to include it. If it is not used, the application will ignore it. See OUTPUT.DOC for output statement syntax.

The Tiger library provides foundation components that are available regardless of the scenario type. There is a document file dedicated to each one.

BITCODE.DOC	A stream that performs NRZ and half rate convolutinal encoding
DEPOT.DOC	A stream that reads generic units from a file
DEVICE.DOC	Defines input and output devices such as files
ERROR.DOC	Components to insert errors into units
EVENT.DOC	Components that define when events occur
FORMATS.DOC	File formats
FRAME.DOC	A generic frame (sync pattern, data, parity)
MUX.DOC	A stream that interleaves multiple streams into one
OUTPUT.DOC	Output components
RECORD.DOC	A stream that makes user-defined units
REGION.DOC	Components to fill units
STREAM.DOC	A generic stream definition
TASK.DOC	Components to perform tasks on units

The following components are also available in any CCSDS application:

PACKET.DOC	A stream that makes version one source packets
PKTCRATE.DOC	A stream that encapsulates units into source packets
TIME.DOC	Additional regions that handle CCSDS time formats

The CADU scenario makes return link telemetry test files. A typical CADU scenario consists of:

a. One or more packet streams. Each packet stream maps to one APID.
b. One mux for each virtual channel that is collecting packets from more than one packet stream. The packet mux interleaves the packets into the virtual channel. You do not need a packet mux if the virtual channel is getting packets from only one packet stream.
c. One or more CADU streams. Each CADU stream maps to one virtual channel. Link each CADU stream to the corresponding packet mux or single packet stream.
d. One mux to interleave the CADUs from the CADU streams. You do not need a virtual channel mux if you have only one CADU stream.
e. One output. Link it to the virtual channel mux or to the single CADU stream.

Do NOT create a CADU or packet stream dedicated to making idle units. Every CADU or packet stream can make idle units as well as regular units, so use existing streams to make idle units.

To write a CADU script, you need the following:

a. This statement:

main c(cadu)

b. One output component. See OUTPUT.DOC.
c. The output needs one or more devices. See DEVICE.DOC.
d. If you are performing NRZ or half rate convolutional encoding, you need a bitcode stream. See BITCODE.DOC.
e. A mux to interleave the CADU streams. See MUX.DOC.
f. One or more CADU streams per virtual channel. See CADU.DOC.
g. A mux to interleave packets for each CADU stream. See MUX.DOC.
h. One or more packet streams per application id. See PACKET.DOC.

Each document file will refer you to other documents to supply additional components. Open the document and copy-and-paste the template. Then change the appropriate fields.

The TC scenario makes forward link telecommanding test files. A typical TC scenario consists of:

a. One packet stream.
b. One TC segment stream if you need it. It breaks packets into segments.
c. One TCTF stream. Link it to the packet or TC segment stream.
d. One CLTU stream. Link it to the TCTF stream.
e. One TC physical stream. It adds the idle and acquisition sequence to each CLTU. Link it to the CLTU stream.
f. One output. Link it to the TC physical stream.

You may create more than one of any of the above streams except output. If you do, then use a mux to interleave their units before you pipe them into the next stream.

To write a TC script, you need the following:

a. This statement:

main c(tc)

b. One output component. See OUTPUT.DOC.
c. The output needs one or more devices. See DEVICE.DOC.
d. One TC physical stream. See TCPHY.DOC.
e. One CLTU stream. See CLTU.DOC.
f. One TCTF stream. See TCTF.DOC.
g. Maybe one TC segment stream. See TCSEG.DOC.
h. One packet stream. Make sure you set the tc(1) argument. See PACKET.DOC.

Each document file will refer you to other documents to supply additional components. Open the document and copy-and-paste the template. Then change the appropriate fields.

4.2 SYNTAX

The script defines which components to build and how to connect them. It contains statements. A statement consists of a path followed by arguments. The following line is a statement example:

```
stream.pkt.a10 appid(10) length(256) max(1000) idleEvent(Idle)
```

The first part, "stream.pkt.a10," is the path and defines a component. In the example, the component is the packet stream named "a10." Paths are hierarchial. The dot separates the levels in the hierarchy.

The fields length(256), max(1000), and idleEvent(Idle) are the arguments. The contents within the parentheses are the argument values.

White space (spaces and tabs) separate paths and arguments; commas do not.

In general, the programs that interpret scripts are unforgiving about mistakes. If they do not understand something, they typically skip it without printing an error. Consequently, do not use white space or punctuation symbols in paths, arguments, or values unless specifically allowed because it could confuse the parser.

Here are general syntax rules:

a. Case is significant. "length(256)" is not the same as "LENGTH(256)."

b. You may write a statement over more than one line by using the continuation character. If the last character (excluding white space) in a line is the "-" character, then the parser will join it with the next non-comment line.

   ```
   stream.pkt.a10 appid(10) - length(256) max(1000) - idleEvent(Idle)
   ```

c. If the first non-white space character in a line is the "#" character, then the parser treats it as a comment line. It ignores blank lines and comment lines. You may not attach comments to statements.

d. Most statements may appear in any order in a script. A few are order dependent. The documention identifies which are order sensitive. The components that the statements define may appear in any order.

e. White space separates arguments from the path and each other. Do not insert white space in paths and arguments.

f. When identifying an argument value, the parser uses "(" and ")" as delimiters. Some arguments permit punctuation and white space in the value, particularly string values. This is the only exception to the rule.

g. The parser does not identify invalid arguments. It uses what it needs and ignores the rest. This means if you mispell or use the wrong case for an optional argument, the parser will not tell you! It will ignore the argument and use this default value. (Someday we will change this.)

h. The last statement in a script may be: *end If you include this optional statement, then the parser treats all statements following it as comments.

The component definitions use the following conventions:

a. The "<" and ">" delimit a string field for which you substitute a name.

b. argument(n) Substitute a decimal value (unsigned unless specified) for the "n." For example, use length(100) for length(n).

c. argument(n:0) The value after the colon is the default value if you omit the argument.

d. argument(1or0) The value may be 1 or 0, which means on or off, true or false.

e. argument(0xnn) The value is in hexadecimal format. For example, fill(0xa1) is a valid substitute for fill(0xnn). If an argument expects a hexadecimal value, then you may omit the "0x" prefix, but it is best to include it for clarity.

f. argument(c) The value is a single character usually selected from a list of characters.

4.3 GLOSSARY

UNIT
A unit is a block of data and the base class for all unit types. Packets, frames, CADUs, V1TFs, CLTUs, et. al. are all units. Tiger's core components all handle units and are unaware of a specific unit type. For example, a mux stream does not know if it is interleaving packets or frames. This means a script-writer could design scripts with unusual linkages. For example, it is possible to link a CADU stream into another CADU stream so that the second CADU stream is putting CADUs into its data zone rather than packets.

IDLE UNIT
An idle unit is a fill unit. Some types of streams can make idle units, such as packets, CADUs, and V1TFs. Others cannot. For example, no telecommanding stream makes idle units. Some idle units have unit id zero.

UNIT ID
Every stream assigns a unit id to every unit that it makes or processes. With one exception, unit ids begin with one and increment upwards, and no two units within a stream have the same unit id. Sometimes a stream will create units with unit id of zero. Unit id zero is always an idle unit, but not all idles have unit id zero. More than one idle unit may have unit id zero.

STREAM
A stream is the general name for a special component that makes or processes units. (Although the output component handles units, it is not a stream.) Most streams make units, but there are also streams to read units from files. The mux stream interleaves units from multiple streams, and the record stream lets user construct custom units. Streams are linked together in scripts, and,

like units, a stream is unaware of the specific stream type to which it is linked. This means a script-writer could design scripts with unusual stream linkages.

OUTPUT
An output is the general name for a special component that gets units from a stream and uses them to write data to one or more output devices. A script will usually have only one output component.

DEVICE
A device is an input or output port. It cannot be both. It is often a disk file, but it can be a network connection, a tape, or any other custom application that derives from the generic device. Outputs write to devices. Streams read from devices. Only one component can use a device; they are not shareable.

REGION
A region is an area within a unit that is defined by a name, a start bit, and a bit length. For example, the version one source packet unit has a data region and an optional secondary header region. The Tiger library provides a palette of patterns from which the script-writer can choose to fill regions. Many application define additional patterns.

TASK
A task is a special operation that streams perform on units. For example, you can use a task to deposit a sequence number anywhere in a unit. Tasks are powerful tools that let you customize units.

ERROR
An error is a special type of task that introduces errors into units. You use an error to simulate instrument and transmission errors.

EVENT
An event tells a stream or output when and how to do something. It associates unit ids with both a number and a true or false state. For example, unit id ten could have value 151 and be true. A common use would be with errors. The event tells the stream to which units it should apply a particular error.

EXPECTED RESULTS FILE
The expected results file is a text file that is a signature of the output test data. There is one expected results file created with every script execution. It contains unit counts, unit maps, and records of errors and gaps.

SCRIPT
The script is a text file that a Tiger application reads to produce test data. The file contains Tiger statements.

4.4 MAIN

The main statement defines the scenario type. If you create a scenario that only uses core components or core components and packets, then use the cadu scenario.

main c(type)

c(type)
The scenario type. Default=cadu.
cadu= CADU scenario.
v1tf= version one transfer frame scenario.
tc= telecommanding scenario.

4.5 CADU STREAM

This stream makes CADUs (frames). It may only be used in the CADU scenario. See STREAM.DOC for general stream information.

The CADU stream can also read CADUs from a file instead of making them. You may also use the depot stream to read CADUs from a file. See DEPOT.DOC.

The stream type must be "cadu." You may choose any name for <name> provided it is different from any other stream name. The usual practice is to use the virtual channel id in the stream name.

```
stream.cadu.<name> service(c) length(n) vcid(n) spid(n) sync(1or0:1) -
        frameSync(0xnnnnnnnn:0x1acffc1d) syncBytes(n:4) max(n:0) -
        replay(ename:0) hdrErrControl(1or0:0) fill(0xnn:0xc9) -
        insertZone(n:0) ocf(1or0:0) insertPDUheader(1or0:1) -
        PDUheaderBitLength(n) PNencode(1or0:0) RSencode(1or0:0) -
        RSinterleave(n:4) RSdual(1or0:1) RScodeLength(n:255) -
        RSempty(1or0:0) crc(1or0) CRCstart(0xnnnn:0xffff) -
        CRCempty(1or0:0) CRCincludeSync(1or0:0) invert(c:N) -
        invertEvent(ename) ERerrors(1or0:1) ERgaps(1or0:1) -
        ERdrop(1or0:1) device(dname) fileType(c) stepSize(n:1) -
        startSequence(n:0) skipZero(1or0:0) drop(ename) -
        idleEvent(ename) idleSeq(1or0:0) reverse(1or0:0)
stream.cadu.<name>.region.data ...
stream.cadu.<name>.region.insertZone ...
stream.cadu.<name>.region.ocf ...
```

The data region is the CADU's data zone. The data region statement is required. The insertZone region statement is required when insertZone(n) is non-zero. It may not be empty (type E). The ocf region statement is required when ocf(1) is set. The region is four bytes long, and it may not be empty (type E). See REGION.DOC for region statement syntax.

This stream may perform tasks. See TASK.DOC. It applies "task" to units before error insertion and encoding, and "xtask" to units after error insertion and encoding.

```
stream.cadu.<name>.task.<task> event(ename) ...
stream.cadu.<name>.xtask.<task> event(ename) ...
```

The following statement shows the mandatory arguments when this stream reads a CADU file and does not make CADUs. When the CADU stream reads CADUs from a file, it ignores all region statements. However, there may be other mandatory arguments if this stream makes idle CADUs.

```
stream.cadu.<name> fileType(c) device(name) length(n)
```

----

service(c)
P= path, encapsulation; V= VCA, VCDU; B= bitstream. Required.

length(n)
Frame length in bytes. Required if crc(1). Min=128. If RSencode(1), the stream computes the frame length and ignores this argument.

vcid(n)
Virtual channel id. Required. Range=0-63.

spid(n)
Spacecraft id. Required. Range=0-255.

sync(1or0:1)
1= prepend frame with sync pattern. Default=1. 0= no sync pattern.

frameSync(0xnnnnnnnn:0x1acffc1d)
Frame sync pattern. Default=0x1acffc1d. The stream puts the four-byte sync pattern before each frame.

syncBytes(n:4)
Number of bytes in sync pattern. Default=4. Must be 0 or 4.

replay(ename:0)
0= real-time, 1= playback. Default=0.
If not 0 or 1, then the argument is a boolean event name. See EVENT.DOC. The stream sets the replay flag to on (playback) when the event is true and off (real-time) otherwise.

hdrErrControl(1or0:0)
1= encode the VCDU header. 0= no encoding. Default=0.

fill(0xnn:0xc9)
Fill byte for idle frames if this stream makes them. Default=0xc9.

insertZone(n:0)
Size of insert zone in bytes. 0= do not use insert zone. Default=0. If you define an insert zone, you must define an insert zone region.

ocf(1or0:0)
Operational Control Field present. Default=0. If you define an OCF, you must define an OCF region. This field is also known as the CLCW.

insertPDUheader(1or0:1)
Meaningful only for services B and P. Default=1.
1= The CADU stream computes the PDU header and inserts it in each CADU. The data region does not include the PDU header.
0= The CADU stream does not compute the PDU header, and the data region includes the PDU header field. You should do this only if the region loader inserts the PDU header such as a raw file might.

PDUheaderBitLength(n)
Number of "good" bitstream data bits. Meaningful for service(B) [bitstream] only and requiredif insertPDUheader(1). The stream uses this number to construct the B_PDU header. if n=0, it deposits the value 16383 (0x3fff) into the header. If n=16383, it deposits 16383. Otherwise, it deposits the value n-1. Note that this value is independent of the data region contents. Furthermore, when the stream fills the data region, it fills the entire region and ignores this value. Also note that it is constant for the entire session. (If you need to vary the bitstream B_PDU length, then use tasks or errors. See TASK.DOC or ERROR.DOC.)

PNencode(1or0:0)
Bit transition density option. Also called pseudo-noise encoding. Default=0. 0= do not encode. 1= encode the CADU. (polynomial: 8-7-5-3-0)

RSencode(1or0:0)
1= Reed-Solomon encoding enabled for frame. Default=0. When on, the stream computes the frame length and overrides the length(n) argument. The following Reed-Solomon arguments are meaningful only when RSencode(1) is set.

RSinterleave(n:4)
Reed-Solomon interleave. Range=1-32. Default=4.

RSdual(1or0:1)
1= assume dual mode. Default=1.

RScodeLength(n:255)
Reed-Solomon codeword length in bytes. Default=255. Range=33-255. If your CADUs have "virtual fill," then subtract the virtual fill count from 255 and set RScodeLength to that value. For example, for RS interleave 1 the virtual fill is three bytes, so set RScodeLength(252).

RSempty(1or0:0)
1= leave space in unit for Reed-Solomon encoding but skip encoding. Default=0. The stream fills the parity area with zeroes. You should set RSempty(1) if the transmitter hardware does RS encoding or if you are only doing a dry run of the script.

crc(1or0)
1= CRC16 encoding enabled for frame. If RSencode(0), then crc is on by default. If REencode(1), then crc is off by default.

CRCstart(0xnnnn:0xffff)
If crc(1), this sets the initial CRC16 encoding value. Default=0xffff.

CRCempty(1or0:0)
1= leave space in unit for CRC encoding but skip encoding. Default=0.

CRCincludeSync(1or0:0)
1= include sync pattern in CRC encoding. Default=0.

invert(c:N)
Invert option. N=none. S=sync pattern only. B=entire frame excluding sync pattern. A=entire frame including sync pattern. Default=none.

invertEvent(ename)
Identifies which frames to invert. Required when invert option is not N. invertEvent(1) means invert every unit. Otherwise it specifies a boolean event. See EVENT.DOC.

idleEvent(ename)
Causes this stream to create idle frames for specified unit ids. The ename value is a boolean event name. See EVENT.DOC.

idleSeq(1or0:0)
This argument tells the stream if it should insert a sequence number in any idle CADUs that it makes. 0= put zero in the sequence field of all idles. 1= put a sequence counter in the sequence field of all idles. Since all idle CADUs are vc 64, the counter is independent of any other sequence counter. Default=0.

reverse(1or0:0)
1= The stream reverses the bit order in every frame. It is the last operation that it performs. 0= forward order frames. Default=0.

See STREAM.DOC for undefined arguments.

4.6 CLTU STREAM

This stream makes CLTUs from TCTFs. It does not put idle or acquisition sequences in CLTUs. It may be used in the TC scenario only. See STREAM.DOC for general stream information.

This stream can read CLTUs from a file instead of making them. You may also use the depot stream to read CLTUs from a file. See DEPOT.DOC. If you use a plain input unit file, then you must supply the sizeBits(n) argument because all CLTUs in a plain unit file must be the same length. This stream cannot read CLTUs from that have idle or acquisition sequences.

The stream type must be "cltu." You may choose any name for <name> provided it is different from any other stream name.

```
stream.cltu.<name> inStream(sname) codeblockLength(n:8) max(n:0) sizeBits(n) -
        ERerrors(1or0:1) ERgaps(1or0:1) drop(ename) ERdrop(1or0:1) -
        device(dlabel) fileType(c)
```

This stream may perform tasks. See TASKS. It applies "task" to units before error insertion and "xtask" to units after error insertion.

```
stream.cltu.<name>.task.<task> event(ename) ...
stream.cltu.<name>.xtask.<task> event(ename) ...
```

----

inStream(sname)
Input stream name. This should be a stream that provides TCTFs. Required unless reading from a file.

codeblockLength(n:8)
Code block length in bytes. Default=8. Range=5-8.

See STREAM.DOC for undefined arguments.

4.7 DEPOT: GENERIC UNIT READER

This stream reads input units from a device using fancy or plain format. See FORMATS.DOC. Many streams are capable of reading input unit files on their own. This stream may replace most of them. It reads generic units. See STREAM.DOC for general stream information.

```
stream.depot.<name> length(n) device(dname) fileType(c) max(n:0) sizeBits(n) -
        ERerrors(1or0:1) ERdrop(1or0:1) drop(ename)
```

This stream may perform tasks. See TASK.DOC. It applies "task" to units before error insertion and encoding, and "xtask" to units after error insertion and encoding.

```
stream.depot.<name>.task.<task> event(ename) ...
stream.depot.<name>.xtask.<task> event(ename) ...
```

----

fileType(c)
Device file type. P=plain unit file. F=fancy unit file. Required.

device(dname)
Device name of device from which this stream will read input units. See DEVICE.DOC. Required.

length(n)
Unit size in bytes. Either length(n) or sizeBits(n) is required.

sizeBits(n)
Unit size in bits. Either length(n) or sizeBits(n) is required.

max(n:0)
The stream will stop producing units when the unit id exceeds the specified id. default=0 (unlimited). Not required. If omitted, there is no limit. This argument may also be called maxUnitId(n).

ERerrors(1or0:1)
1= put unit error information into the expected results file. default=1.

ERdrop(1or0:1)
1= identify discarded units in the expected results file. default=1.

drop(ename)
This argument causes the stream to discard units based upon the event. See EVENT.DOC. If omitted, the stream does not drop any units. Not required. If present, the event must exist in the script.

See STREAM.DOC for undefined arguments.

4.8 DEVICES

A device defines a specific input or output target. It is often a file, but it could be a port, a tape, or null. Other components, such as streams and outputs, link to devices through the device name for either input or output. A component exclusively owns a device. Devices are never shared.

The general format for a device is:
device.<type>.<dname> <arguments>

Currently there are only two device types: null and file. Other applications may define additional types. The file device type is usually used for devices such as disk files. The null device type is a pseudo-port because all output is discarded and all input is simulated. It is useful when testing a new script.

You choose the device name (dname). It must be unique among devices. Components reference the device name to link to the device.

FILE DEVICE

device.file.<dname> name(filename) access(c)

name(filename)
The complete file name string. Required.

access(c)
Access mode. r= read, w= write. Required.

NULL DEVICE

When used as an output device (access(w)), the script discards all output but still reports on the quantity of data discarded. When used as an input device (access(r)), the script simulates input but does not actually read a file. The script needs the file length to determine when to stop processing input.

device.null.<dname> access(c) length(n)

access(c)
Access mode. r= read, w= write. Required.

length(n)
Number of bytes in an input device. Required if access(r).

4.9 ERRORS

An error statement instructs a stream to introduce errors into units. Error statements are always attached to streams. Each error requires an event to determine when the stream should apply an error. The error types are:

flip	**invert a consecutive string of bits.**
set	**deposit a 1-32 bit value somewhere in a unit.**
add	**add a 1-32 bit value somewhere in a unit.**
flipmask	**flip up to 32 bits corresponding to "on" bits in a 32-bit mask.**
resize	**truncate or extend a unit**

The general formats are:

stream.<stype>.<stream>.error.flip convey(1or0:0) label(text) -
event(ename) startbit(n) bits(n)

stream.<stype>.<stream>.error.set convey(1or0:0) label(text) -
event(ename) startbit(n) bits(n) v(n)

stream.<stype>.<stream>.error.add convey(1or0:0) label(text) -
event(ename) startbit(n) bits(n) v(n)

stream.<stype>.<stream>.error.flipmask convey(1or0:0) label(text) -
event(ename) startbit(n) bits(n) emask(0xnnnnnnnn)

stream.<stype>.<stream>.error.resize convey(1or0:0) label(text) -
event(ename) chop(1or0) fill(0xnn:0) v(n) bits(1or0:0) abs(1or0:0)

The <stype> and <stream> paths are respectively the type and name of the stream to which the error is attached. The resize error is different from the other types, so its description follows the following section.

label(text)
This text is printed in the expected results file when the stream applies the error. It helps to identify errors. Choose descriptive text. Required.

event(ename)
This argument links the error to an event, which tells the stream which units will have the error applied to them. The ename value is an event name. See EVENT.DOC. If you specify event(1), then the stream applies the error to every unit.

startbit(n)
The startbit and bits arguments define a location in a unit where the error is applied. Startbit may be positive or negative. When zero or positive, the stream measures the location from the start of the unit with the first bit being bit zero. The maximum value for startbit is the unit size in bits minus one. When starbit is negative, the stream measures the location from the end of the unit. For example, startbit(-1) points to the last bit, and startbit(-16) points to 16 bits from the unit end. Required.

bits(n)
The number of bis in the error field. See startbit(n) description. For flip, it may be as large as the unit size in bits minus startbit(n). If you set bits(0), the stream inverts all bits in the unit starting with startbit(n) to the end of the unit. For all other error types, the range is 1 to 32 bits. Required.

convey(1or0:0)
1= The error conveys to all receiving streams. A receiving stream will mark a unit as having errors if any part of it contains fragments from a unit with conveyed errors. If a stream is encoding a unit, the error is applied AFTER the encoding is done. When using resize, you should almost always set convey(1).
0= The stream hides the error from a receiving stream. Default=0. If a stream is encoding a unit, the error is applied BEFORE the encoding is done. See the discussion at the end of this section for more information.

v(n)
This is a value that is added or deposited into a unit. If omitted, the add and set errors use the value associated with the event. This argument may also be specified as value(n).

emask(0xnnnnnnnn)
This mask identifies which bits to flip in the field. "On" bits are flipped. Required.

----
stream.<stype>.<name>.error.resize convey(1or0:0) label(text) event(ename) -
chop(1or0) fill(0xnn:0) v(n) bits(1or0:0) abs(1or0:0)

label(text)
This text is printed in the expected results file when the stream applies the error. It helps to identify errors. Choose descriptive text. Required.

event(ename)
This argument links the error to an event, which tells the stream which units will have the error applied to them. The ename value is an event name. See EVENT.DOC. If you specify event(1), then the stream applies the error to every unit.

convey(1or0:0)
1= The error conveys to all receiving streams. See explanation above. You should almost always set convey(1) for resize because you almost always want to do encoding, such as CRC or RS, before you truncate or extend a unit. Doing it before encoding may cause unexpected results and even crashes for certain unit types such as frames.
0= Truncate before encoding. The error is hidden from subsequent streams. Default=0.

v(n)
The meaning of this value depends on other arguments. It is either the number of bits/bytes to extend or truncate a unit, or it is the desired size of the resized unit. If omitted, resize gets the value from the associated event.

chop(1or0)
1= truncate the unit. 0= extend the unit. You cannot do both with the same error. Required.

fill(0xnn:0)
When chop(0) and you are extending a unit, this is the fill byte for the extended area.

bits(1or0:0)
Truncate by bits or bytes. 1= v(n) is in bits. 0= v(n) is bytes. Default=0.

abs(1or0:0)

This flag tells the stream if v(n) is the total number of bits/bytes in the final unit or if v(n) is the number of bits/bytes to truncate or extend the unit. Default=0. 0= v(n) is the number of bits or bytes to truncate or extend the unit. 1= v(n) is the target total number of bits or bytes in the unit.

CONVEYING ERRORS

When you set convey(1) in an error statement, the stream applies the error after it has done any encoding. Furthermore, the stream propagates the error to every receiving stream. Each receiving stream applies the error after it has constructed its own unit and after it has done any encoding.

For example, suppose a script has a packet stream and a frame stream. The frame stream is making frames from the packets it gets from the packet stream. If we introduce an error to a packet and we set convey(1) in the packet stream, then the packet error is conveyed to the frame stream. The frame stream will build a frame from a clean, error-free copy of our bad packet. It will then encode the clean frame, and only then will it apply the packet error. This means if the frame stream is doing CRC or Reed-Solomon encoding, the frame will have a CRC or Reed-Solomon error because the error is applied after the encoding.

When you set convey(0), the stream applies the error before it does any encoding. Furthermore, it hides the error from any receiving stream. If we set convey(0) in our frame and packet example, then the frame stream would encode the frame over the top of the bad packet. This means if it's doing CRC or Reed-Solomon encoding, the frame will not have a CRC or Reed-Solomon error. In this case, convey(0) simulates an instrument error rather than a transmission error.

Usually when you put errors into packets, you are testing equipment to see how it handles particular packet field errors. Under these circumstances, you should probably set convey(0) in your packet streams. If you set convey(1) and the target equipment has Reed-Solomon correction hardware, then that hardware will most likely correct your errors at the frame level, and your carefully planned errors will be lost. If you want to test the Reed-Solomon correction hardware, it is best to insert the errors in the frame stream and to set convey(1) to ensure the errors are applied after encoding.

4.10 EVENTS

An event causes something to happen for a specific unit id. For example, a packet stream uses an event to determine which packets should have errors. An event also associates a value with a unit id. For example, a packet stream, which makes variable length packets, uses an event to determine the length of each packet. More than one component may use the same event.

An event is a collection of event statements with the same event name and type. The general format is:

```
event.<type>.<ename> default(n)
event.<type>.<ename>.range uid0(n) uid1(n) v(n)
event.<type>.<ename>.recur start(n) repeat(n:0) skip(n:0) span(n:0) -
            occur(n:0) v(n)
```

The general format shows three event statements, but not all are required. An event may have zero or more range or recur statements in any combination. It may or may not have one default statement, which is determined by the event type and context. You choose the event name (ename). It must be unique among events. Components reference the event name to link to the event. For type, use "value" or "unit."

There are two event types: value and unit. A value event has both a true/false state and a numeric value associated with every unit id. A unit event (also called a boolean event) has a true/false state for every unit id and only a trivial unit id value (1 for true and 0 for false). Streams ask events to return either a value or a true/false state for each unit id depending on the context. For example, a packet stream might need to know the packet length for packet 10. It would ask a value event to return the value for unit id 10, which it would use as the packet length. A frame stream might need to know if it should make frame 60 as an idle frame. It would ask a unit event or a value event to return the true/false state for unit id 60. Typically, there is a stream statement argument that links a stream action to an event name. For example, the packet stream has varLenEvent(ename) and the frame stream has idle(ename) as statement arguments. The "ename" value is an event name, which points to a particular event in the script.

Here is an example of a value event:

```
event.value.ev1.range uid0(1) uid1(10) v(1)
event.value.ev1.range uid0(100) uid1(150) v(2)
event.value.ev1.recur start(1) skip(1) repeat(0) v(5)
event.value.ev1 default(13)
```

The event is referenced by the name "ev1." Unit ids 1-10 have value 1, unit ids 100-150 have value 2, odd unit ids not in the ranges have value 5, and all others have value 13. Unit ids 1-10, 100-150, and all odd ids are true, and the rest are false.

Here is an example of a unit (boolean) event:

```
event.unit.ev1.range uid0(1) uid1(10)
event.unit.ev1.range uid0(100) uid1(150)
event.unit.ev1.recur start(1) skip(1) repeat(0)
```

In the example, unit id 1-10, 100-150, and all odd unit ids are true. The rest are false. All true units have value 1, and the rest have value 0. When an argument wants a unit event, you may also point it to a value event. The stream will ignore the values and only look at the true or false state.

RECURRENT PATTERN STATEMENT

The recur statement defines a repeating pattern of units. For example, you would use the recur statement to give every odd unit id a true state.

A recurrent pattern might look like this: tttt__tttt__tttt__tttt__ In this example, unit ids 1-4, 7-10, 13-16, and 19-22 are true, and all others are false. For this pattern example, the script line would be either of the follow two:

```
event.unit.<name>.recur start(1) repeat(3) occur(4) skip(2)
event.unit.<name>.recur start(1) repeat(3) occur(4) span(6)
```

or one of the following for a value event with a true value=25 and false=0:

```
event.value.<name> default(0)
event.value.<name>.recur start(1) repeat(3) occur(4) skip(2) v(25)
event.value.<name>.recur start(1) repeat(3) occur(4) span(6) v(25)
```

The complete format for a recurrent event state is:

```
event.<type>.<name>.recur start(n) repeat(n:0) skip(n:0) span(n:0) -
          occur(n:0) v(n)
```

start(n)
The starting unit id. Required. Minimum=1.

repeat(n:0)
Number of times to repeat the true state in a group. It is the number of consecutive trues minus one. default=0.

skip(n:0)
Number of false values between groups of true values. Default=0.

span(n:0)
Number of units in pattern. span=repeat+skip+1. If span(n) is present, skip(n) is ignored and computed as span-repeat-1. Default=0 (span is computed.) If present, the minimum value is repeat+1. In practice, it is often much simpler to supply span(n) then to compute skip(n).

occur(n:0)
Number of occurrences of the pattern. default=0, which is infinite.

v(n)
Value associated with the state. Required for value events.

RANGE STATEMENT

A range defines an interval by a start and end unit id. A unit id is true if it lies within the interval, including the end points.

```
event.unit.<name>.range uid0(n) uid1(n)
event.value.<name>.range uid0(n) uid1(n) v(n)
```

uid0(n)
First unit id of range. minimum=1. Required.

uid1(n)
Last unit id of range. minimum=uid(0). Required.

v(n)
Value associated with the state. Required for value events.

To specify the false value for a value event, you must specify the default(n) argument as in the following example:

```
event.value.ev1.range uid0(1) uid1(10) v(1)
event.value.ev1.range uid0(100) uid1(150) v(2)
event.value.ev1.recur start(1) skip(1) repeat(0) v(5)
event.value.ev1 default(13)
```

The default argument is not always required; it depends on the usage. For example, if the event is used to determine idle units, then default(n) is not used because the stream creates idles only for trues. However, if the same event is used for packet lengths, then it is required because the stream uses the event for all unit ids, and the true or false state is not important. There is no default for default(n).

Different event states may have overlapping unit ids. To resolve an event (especially when the states have values and overlapping unit ids), Tiger always searches through all range event states before it searches through the recurrent event states, regardless of the listed order. If there is more than one event state of the same type (range or recur), then Tiger searches through them in the order they are listed in the script. Tiger stops searching as soon as it finds a true. For example,

```
event.value.evx.recur start(1) skip(1) v(3)
event.value.evx.range uid0(1) uid1(20) v(1)
event.value.evx.range uid0(15) uid1(20) v(2)
```

The value for unit id 17 is 1, despite the fact that all three states reference id=17, because it satisfies the first listed range event.

4.11 FORMATS

PLAIN FILE FORMAT

The plain file format is a continuous stream of units without headers, trailers, or markers of any kind. The file is binary and not text. The plain output component writes units to the file in the order that the units are produced.

FANCY FILE FORMAT

The fancy file format is a Tiger invention. The file has a header, and each unit has one too. The file is binary and not text. Like the plain format, it contains a continuous stream of units but with annotation. This format is most suitable for making intermediate data files because no pertinent information is lost. For example, one could run a script that creates a fancy file of packets. In the next stage, one could run a frame generation script that uses the packet file without information loss.

It is possible to use a plain file for staging but with limitations. Plain files do not contain unit information, so, for example, it is impossible to tell which packets have errors in a plain file. This means it is impossible to convey errors to a frame stream from a plain packet file.

Furthermore, the plain unit file must contain fixed size units. Tiger cannot read a plain unit file with variable size units because it does not know where the units begin and end. You must use a fancy file format to create intermediate files for variable size units.

File Header
The fancy file header contains a fixed length primary header and an optional secondary header. The tccsds program does not create or use the secondary header, but special applications or later versions may use it. Applications should not assume the secondary header is empty but instead should use the secondary header length field, which is in the primary header, to find the start of data.

If you are using the files output component with the fancy format, note the that fancy header is placed on the first file only and not every subsequent file. (The files component splits a large output file into a collection of smaller files.) The files component does not split units between files.

Primary Header
The primary header is 20 bytes. Each field is four bytes long.

version	The fancy file format version number. This document describes version 1. Applications should check the version number and may abort or warn the user if the version is greater than they expect.
fancyVerification	This field contains the pattern 0xa495. Applications should use it to verify that the data file is truly a fancy format file.
secondaryHeaderLength	The secondary header length in bytes.
secondaryAnnotationLength	Unit secondary annotation length in bytes.
annotationLength	Unit primary annotation length in bytes.

The units immediately follow the file header. Each unit also has a header (annotation). As with the file header, this is a fixed length primary annotation and an optional secondary annotation. The secondary annotation length does not vary from unit to unit, and its length is establish in the file's primary header. The tccsds program does not create or use the secondary annotation, but special applications or later versions may use it. Applications should not assume the secondary annotation is empty but instead should use the secondary annotation length field, which is in the primary header.

Primary Annotation

The primary annotation is fixed length. Later versions may append new fields to it, so applications should use the length that is specified in the file primary header. (Applications should also use the secondary annotation length from the primary header to find the beginning of data.)

4 bytes	unitId	Unit id. This is an incrementing unit number starting with 1. There occasionally may be a unit with id=0 in the file. These are always idle units, but not all idles have id=0.
4 bytes	bitLength	Unit length in bits
4 bytes	byteLength	Unit length in bytes rounded up to the next byte for odd bit size units
4 bytes	streamId	Stream id of creator stream
1 byte	conveyError	1= The unit is conveying errors
1 byte	hasErrors	1= The unit has errors
1 byte	hasTwoPunits	1= The unit has two physical units (see below)
1 byte	hasConstantFill	1= The unit contains constant-pattern fill data
1 byte	isIdle	1= The unit is an idle (fill) unit
3 bytes	spare	Not used

The secondary annotation, if it exists, immediately follows the primary annotation.

The unit data immediately follows the secondary annotation. Warning! There may be TWO COPIES of the unit data! If the "hasTwoPunits" flag in the primary annotation is set, then there will be two back-to-back copies of the unit data. The first copy will contain errors, and the second copy will be without errors.

The only time you should expect to see two copies is when a unit has errors and is conveying them. The reason two copies is necessary, a bad one and a clean one, is that streams that are encoding their units need the clean copy to correctly encode them.

SIM FORMAT

The SIM format is designed for a GSFC Code 520 simulator board and SIM transmitter software. The SIM format consists of two files: a base file and an update file. The board contains a fixed capacity of memory (usually four megabytes), which limits the size of the base file. The update file handles units which extend beyond the size of the SIM memory.

The base file contains a 128-byte header immediately followed by the unit data. The header is not loaded into the board's memory, so it does not count against the memory limitation. The units are not annotated. The files are binary format.

SIM Base File Header (128 bytes)

2 bytes	pattern	This pattern is always 0xad00.
2 bytes	records	Number of records in the base file.
4 bytes	recordSize	Size of each record in bytes.
4 bytes	memory	Bytes of memory used up to the maximum. This will be records times recordSize.

4 bytes	offset	Byte offset to first record in memory
4 bytes	memorySize	Number of kilobytes of memory on the board.
108	notUsed	Not used.

A record is usually a unit, but it is not required. You might use the record stream to create artificial units of any size. You would need to do this if your units were variable length or if you wanted to transmit a text file. The record size must be a multiple of four bytes.

The update file does not contain units. It contains the changes that are needed to be made to the records currently in memory to bring them up to date. In other words, the update file is a file of differences. The update file is a stream of variable length update blocks. Each block consists of a 32-bit address and count register followed one or more 32-bit fields of actual update data.

1 byte	count	Number of 32-bit update fields in this block. One block will update from one to 254 consecutive 32-bit memory locations.
3 bytes	address	Location in memory to update. The SIM memory is treated as an array of 32-bit integers such that address zero is the first 32 bits of memory. In effect, the address is a long array index. The update hardware pastes the update data into memory starting at this location.
1-254 longs	data	The update data.

STGEN FORMAT

The STGEN format is designed for a GSFC code 520 simulator board and STGEN transmitter software. The board contains a fixed capacity of memory (usually four megabytes), so STGEN files must be less than or equal to that size. The file contains a 128-byte header immediately followed by the unit data. The header is not loaded into the board's memory, so it does not count against the memory limitation. The units are not annotated. The file is binary format.

STGEN Header (128 bytes)

2 bytes	pattern	This pattern is always 0xff00.
2 bytes	records	Number of records in the file
4 bytes	recordSize	Size of each record in bytes
4 bytes	memory	Bytes of memory used up to the maximum. This will be records times recordSize.
116	notUsed	Not used.

A record is usually a unit, but it is not required. You might use the record stream to create artificial units of any size. You would need to do this if your units were variable length or if you wanted to transmit a text file.

TDG FORMAT (TPGEN)

The TDG format is designed for a GSFC code 520 simulator board and TDG transmitter software. The board contains a fixed capacity of memory (usually four megabytes), so TDG files must be less than or equal to that size. The file is binary format.

The TDG format is similar to the STGEN format but with severe limitations. There are at least two variations in the format. Consequently, we do not present the format here, primarily because we discourage anyone from continuing to support it. If you must know the format, see the transmitter software and its documentation.

4.12 MUX

The mux stream interleaves multiple input streams into one output stream. It does not support tasks or errors. See STREAM.DOC for general stream information.

The stream type must be "mux." You may choose any name for <name> provided it is different from any other stream name.

The mux stream uses one of two strategies to interleave units. With strategy A, the mux determines unit order by using order statements, which resemble an event. With strategy F, the mux reads a text file, which contains the unit interleave order by stream name. The general format is:

```
#STRATEGY A (DEFAULT)
stream.mux.<name> strategy(A) default(sname) idle(sname) max(n:0) -
      eos(c:S) er(ename:0) erSize(1or0:0)
stream.mux.<name>.range stream(sname) uid0(n) uid1(n)
stream.mux.<name>.recur stream(sname) start(n) repeat(n:0) skip(n:0) -
      span(n:0) occur(n:0)
stream.mux.<name>.range idle(n) uid0(n) uid1(n)
stream.mux.<name>.recur idle(n) start(n) repeat(n:0) skip(n:0) -
      span(n:0) occur(n:0)

#STRATEGY F (USING A FILE)
stream.mux.<name> strategy(F) default(sname) idle(sname) max(n:0) -
      eos(c:S) er(ename:0) erSize(1or0:0) file(filename) idleBitSize(n)
```

Under strategy A, you may specify any number of range and recur statements in any combination. The syntax is nearly identical to that of events. See below for more information about the strategies.

strategy(c:A)
Interleave strategy. A=use recur and range statements to determine order. F=read order from file. Default=A.

default(sname)
The default stream is the backup stream. The mux uses it to get a unit whenever the selected stream cannot provide one. This can happen if the selected stream stops giving units or if the unit id is not linked to a stream. The mux stops if the default stream stops providing units. This argument is optional. The mux will stop if it needs a unit from the default stream and you omitted the default(sname) argument.

max(n:0)
This is the maximum number of units that this mux will process. The mux stops when the number is processed. default=0 (unlimited).

idle(sname)
Identifies the input stream that provides idle units. You may omit this argument. However, the mux will then abort the scenario if it is asked to provide an idle unit or it is programmed to generate idle units. For CCSDS CADU and V1TF scenarios, any packet or frame stream can service idle unit requests. You should omit this argument for telecommanding (TC) scenarios because there are no idle units in telecommanding streams.

idleBitSize(n)
Size of idle unit in bits. Only used under strategy F. It is required if the mux file contains an idle unit declaration and the input stream does not always make a fixed size unit, such as packets.

eos(c:S)
What to do when the selected input stream returns no unit. default=S.
S= skip the stream and use the default stream instead. If the default stream has stopped, then the mux stops.
I= get an idle idle from that stream. This only works if the input stream makes fixed size units because the mux requests zero bytes as the idle unit size.
X= stop the mux.

er(ename:0)
Determines if and when to print information to the expected results file. er(1)= always. er(0)= never. Default=0. You may also specify a unit event, and the mux will print expected only for true units. See EVENT.DOC. This option writes stream and unit ids for all units that pass through the mux.
erSize(1or0:0)
This argument is meaningful only if er(1) or er(ename) is set.
1= write unit size (bytes) to the expected results file. Default=0. You should only turn it on if the mux is handling variable size units.

file(filename)
Text file that contains the interleave order. Required for strategy F and not used for strategy A.

MUX INTERLEAVE STRATEGY A

You must provide one or more of the following range or recur statements to identify the list of input streams and the sequence in which the mux uses them. The mux uses its unit id as a key. If the unit id is not in any list, the mux uses the default stream. If the default stream runs out of units, the mux stops. See EVENT.DOC for range and recur statement syntax.

```
stream.mux.<name>.range stream(sname) uid0(n) uid1(n)
stream.mux.<name>.recur stream(sname) start(n) repeat(n:0) -
     skip(n:0) span(n:0) occur(n:0)

stream.mux.<name>.range idle(n) uid0(n) uid1(n)
stream.mux.<name>.recur idle(n) start(n) repeat(n:0) skip(n:0) -
     span(n:0) occur(n:0)
```

stream(sname)
Input stream name. This argument is ignored if idle(n) is specified. Either idle(n) or stream(sname) is required.

idle(n)
If present, the mux requests idle units from the idle stream of length n BITS. Set idle(0) if the idle stream makes fixed size idles. When idle(n) is present, the mux ignores the stream(name) argument.

MUX INTERLEAVE STRATEGY F

The mux gets the interleave order from a text file. The file consists of a lists of stream names, one name per line. The mux request each stream in order for one unit. When it reaches EOF, it begins the file again from the beginning. Left justify each name, and do not include internal or trailing whitespace or blank lines. Here is an example:

```
ap1
ap1
ap2
idle
```

ap3
ap2
ap1

The name "idle" is special and indicates that the mux should request an idle unit. This means the name "idle" is reserved, so you should not use it as a stream name. When you use idle, you may need to use idleBitSize(n) in the main mux statement.

```
# The muxtool makes a mux input interleave file (strategy F). Some users
# wish to have certain streams provide units by percentage. For example,
# you might want to have "vc17" frames only appear in 12% of the created
# frames. By using muxtool with stream names and percentages or unit
# counts, you can create complicated interleave files. You may edit these
# files to add or subtract as you need. muxtool will attempt to evenly
# distribute the streams over the requested span.

# The first column is a list of stream names. The second column is a list
# of unit counts. If you use a stream name of "idle" as in the example,
# the mux will request idle units. Do name any stream "idle" in your
# scripts!

# The syntax is: muxtool <thisfile> [<span:100>]. The span is the
# number of units (lines) in the mux interleave file. muxtool scales the
# second column so that the new total is equal to span (or nearly so).
# The default value is 100. The interleave file will be the input file
# plus ".mux" extension. The second column does not need to aligned or
# in a particular column. Just separate the number from the name by blank
# space. You may run this doc file through muxtool as a test. Blank lines
# and lines beginning with "#" are ignored.
```

stream	10	10
stream	25	25
stream	1	1
stream	5	5
stream	50	50
stream	45	45
stream	12	12
stream	2	2
stream	55	55
stream	43	43
idle	1	

4.13 OUTPUT

Output describes what kind of output products the script will produce. You must define one output in every script. (Some Tiger applications may permit or require more than one ouput. Check the documentation.) Every output connects to one stream, from which it reads units (packets, frames, etc). Every output connects to at least one device to which it writes the final output products.

The general format for an output statement is:
  output.<otype> <arguments>

Substitute an output type label for <otype>. The available output types, which the tccsds program will create, are:

  plain
  fancy

files
sim
stgen
tdg

Other Tiger applications may restrict or add to the list of available output types. Check the documentation.

Almost all outputs share three common arguments. The inStream(sname) argument tells the output which stream will provide its input units. The argument value is a stream name, so you must define one stream by that name in the script. The max(n:0) is an optional argument that tells the output how many input units to process before stopping. If an input stream stops producing units, then the output will stop regardless of the value of max argument. The device(dname) argument is the device to which the output writes the product. Most outputs require that you define at least one device statement in the script.

PLAIN OUTPUT

This output type writes units to a device without headers or trailers. It demands one device for output and one stream for input. The output is usually a binary file.

output.plain device(dname) inStream(sname) max(n:0)

device(dname)
Output device name. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

inStream(sname)
Input stream name. Required. You must include a stream definition by that name in the script. See the documentation for the stream type that you want to plug into this output.

max(n:0)
Maximum number of units to write to the device. 0=unlimited. Default=0.

FANCY OUTPUT

This output type writes units to a device using a special Tiger format. It demands one device for output and one stream for input. The output is usually a binary file. This format is the best choice if you plan to run a scenario in stages and you want to create temporary, intermediate data files. For example, in one script you create a file of packets. Later you run another script to create frames that uses the packet data file as input. It is best in this case to make the packet data file using the fancy output because that format retains all of the necessary per unit information. See FORMATS.DOC for file format details.

output.fancy device(dname) inStream(sname) max(n:0)

device(dname)
Output device name. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

inStream(sname)
Input stream name. Required. You must include a stream definition by that name in the script. See the documentation for the stream type that you want to plug into this output.

max(n:0)
Maximum number of units to write to the device. 0=unlimited. Default=0.

FILES OUTPUT

This output type enhances the plain and fancy formats. It creates a set of smaller output files in place of one huge output file.

output.files device(dname) inStream(sname) max(n:0) KBperFile(n) type(c)

device(dname)
Output device name. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

inStream(sname)
Input stream name. Required. You must include a stream definition by that name in the script. See the documentation for the stream type that you want to plug into this output.

max(n:0)
Maximum number of units to write to the device. 0=unlimited. Default=0.

type(c)
File type. Required. For plain format, c=plain or P. For fancy format, c=fancy or F.

KBperFile(n)
Number of kilobytes to place into each file. Required. It does not split units between files. It appends a sequence number to each file name to distinguish them. For example, if the output file is chunk.dat, then it creates chunk.dat, chunk.dat.0, chunk.dat.1, chunk.dat.2, etc. The fancy format will almost always exceed KBperFile by as much as one record, so make sure you reserve some space for this information (at least one full record). For your information, 1 gigabyte = KBperFile(1048576).

SIM OUTPUT

This output type creates files to support the GSFC code 520 SIM card. It consists of a base file and an update file, so you must specify two devices instead of the normal one device. All units from the input stream must be the same size. (If they are not, then you must pipe the input stream into a record, which you then attach to the output. See RECORD.DOC.)

output.sim inStream(sname) base(dname) update(dname) memorySizeMB(n:4) -
        recsPerSide(n) recordSize(n) max(n:0)

inStream(sname)
Input stream name. Required. You must include a stream definition by that name in the script. See the documentation for the stream type that you want to plug into this output.

base(dname)
Device name for the base file. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

update(dname)
Device name for the update file. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

memorySizeMB(n:4)
Number of megabytes in the entire SIM card memory. Default=4.

recsPerSide(n)

Number of records per side. The SIM memory is divided into an A and B side. If you omit this argument, it fills both sides with as many units as possible. The scenario will fail if the input stream does not provide not enough data to fill the specified memory on the SIM card.

max(n:0)
Maximum number of units to write to the devices. 0=unlimited. Default=0. The minimum non-zero value is two times recsPerSide.

recordSize(n)
Number of bytes in each record (unit). It must be divisible by four. Required. This is usually the size of the input unit.

STGEN OUTPUT

This output type creates a file to support the GSFC Code 520 SIM card that uses the STGEN data format. It demands one device for output and one stream for input. All units from the input stream must be the same size. (If they are not, then you must pipe the input stream into a record, which you then attach to the output. See RECORD.DOC.) See FORMATS.DOC for the file format.

output.stgen inStream(sname) device(dname) memorySizeMB(n:4) recordSize(n) -
    max(n:0)

inStream(sname)
Input stream name. Required. You must include a stream definition by that name in the script. See the documentation for the stream type that you want to plug into this output.

device(dname)
Output device name. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

recordSize(n)
Number of bytes in each record (unit). Required. This is usually the size of the input unit.

memorySizeMB(n:4)
Number of megabytes in the SIM card memory. Default=4.

max(n:0)
Number of records (units) to write to the device. The default behavior is to fill the memory with as many units as possible. The output stops as soon as it fills memory regardless of the value of max(n).

TDG OUTPUT

This output type creates a file to support the GSFC code 520 SIM card that uses the TPGEN/TDG data format. It demands one device for output and one stream for input. All units from the input stream must be the same size. (If they are not, then you must pipe the input stream into a record, which you then attach to the output. See RECORD.DOC.)

output.tdg inStream(sname) device(dname) memorySizeMB(n:4) recordSize(n) -
    max(n:0) updates(1or0:1)

inStream(sname)
Input stream name. Required. You must include a stream definition by that name in the script. See the documentation for the stream type that you want to plug into this output.

device(dname)

Output device name. Required. You must include a device.<dname> statement in the script. See DEVICE.DOC.

recordSize(n)
Number of bytes in each record (unit). Required. This is usually the size of the input unit.

memorySizeMB(n:4)
Number of megaobytes in the SIM card memory. Default=4.

max(n:0)
Number of records (units) to write to the device. The default behavior is to fill the memory with as many units as possible. The output stops as soon as it fills memory regardless of the value of max(n).

updates(1or0:1)
Determines which style of TDG file to create. 1= create file with updates. 0= create file without updates. Default=1. If you plan to use a TDG debugger to transmit test data, use the updates(0) option. Otherwise, try the updates(1) option.

4.14 PACKET STREAM

This stream makes version one source packets. It can be used in any CCSDS scenario. See STREAM.DOC for general stream information.

This stream can also read packets from a file instead of making them. You may also use the depot stream to read packets from a file. See DEPOT.DOC.

The stream type must be "pkt." You may choose any name for <name> provided it is different from any other stream name. The usual practice is to use the application id in the stream name.

The data region is the packet's data zone. The data region statement is required. The 2hdr region refers to the secondary header. It is required if 2hdr() is set. Often, time is inserted in the secondary header. See REGION.DOC for region statement syntax. See TIME.DOC for time patterns.

This stream may perform tasks. See TASK.DOC for sytax. It applies a task after filling regions but before error insertion. The task path name is "task."

```
stream.pkt.<name>.task.<task> event(ename) ...
```

The following statement shows the mandatory arguments when this stream reads a packet file and does not make packets. When the packet stream reads packets from a file, it ignores all region statements. However, the stream may still use time segments, it may still apply errors and tasks, and it may create idle packets.

```
stream.pkt.<name> fileType(type) device(name) length(n) appid(n)
```

```
--------------------
stream.pkt.<name> appid(n) tc(1or0:0) variableLength(1or0:0) length(n) -
           fill(0xnn:0xc9) varLenEvent(ename) 2hdr(ename:0) -
           2hdrLength(n) version(n:0) checksum(1or0:0) dataSeq(1or0:0) -
           idleEvent(ename) max(n:0) ERerrors(1or0:1) ERtseg(1or0:1) -
           ERgaps(1or0:1) ERdrop(1or0:1) drop(ename) device(dname) -
           fileType(type) seq(1or0:1) stepSize(n:1) startSequence(n:0) -
          skipZero(1or0:0)
stream.pkt.<name>.region.data ...
stream.pkt.<name>.region.2hdr ...
```

appid(n)

Application id. Required. Range=0-2047. If reading packets from a file, this number appears in the expected results report, and the stream does not insert it into any packets. It does put it into the primary header when it is making packets.

tc(1or0:0)
1= telecommanding packet. 0=telemetry packet. Default=0. When using the telecommanding scenario, set tc(1).

variableLength(1or0:0)
1= variable length packets, 0= fixed length packets. Default=0.

varLenEvent(ename)
Name of value event that associates unit ids with packet byte lengths. See EVENT.DOC. Meaningful only if variableLength(1) is set. The event values are packet byte lengths. Not Required. See below.

length(n)
Packet length in bytes. Required if variableLength(0). Min=7. If this packet stream is reading packets from a file using fileType(F) or fileType(VPKT), then it gets each packet length from the file. Despite this, you must still provide a valid length(n) argument even though the stream does not use length(n) to process packets.

2hdr(ename:0)
Determines if a secondary header is present. 1= secondary header present. 0= no secondary header. Default=0. Otherwise, the argument is a unit event name. See EVENT.DOC. The stream inserts a secondary header whenever the event is true, and it omits the header when the event is false. If the secondary header option is on, you must specify 2hdrLength(n), and you must define a 2hdr region.

2hdrLength(n)
Secondary header length in bytes. Required if 2hdr(1).

version(n:0)
Packet version number. Default=0. Range=0-7.

checksum(1or0:0)
1= put checksum (sum of all previous bytes modulo 256) in last byte of packet. Default=0.

dataSeq(1or0:0)
1= Put the unit id into the first 32 bits of the packet data region, which immediately follows the secondary header. It overwrites any pattern in that area and becomes part of the data. This feature can be useful in verifying packet reassembly because it puts a non-wrapping counter in the packet. Note that the packet must have at least 32 free data bits to enable this option. Default=0.

fill(0xnn:0xc9)
Used for data in idle packets. Default=0xc9.

idleEvent(ename)
Causes this packet stream to create idle packets for specified unit ids. The ename value is a value event name. See below and also see EVENT.DOC. The values are byte lengths. Do not link to events with values that are less than seven bytes.

seq(1or0:1)
1= put sequence number in packets. 0= set sequence field to zero in every packet. Default=1.

ERtseg(1or0:1)
1= write time segment information to the expected results file. Default=1. This argument is meaningful only if this stream has tseg statements.

fileType(type)
Device file type. P=plain unit file. F=fancy unit file. VPKT=packet file. It is required if device(dname) is specified. The VPKT type is specific to this stream. When used, the packet stream reads packets from a plain file of packets. It determines each packet length by getting the length field from each packet primary header. The packet file has no special characteristics and is simply a plain unit file of type(P). The fileType(VPKT)type allows the stream to read a plain file of variable length packets, which is not possible under fileType(P). The packet lengths in the file must be true and without errors.

See STREAM.DOC for undefined arguments.

GROUP FLAGS

The packet header contains a pair of bit flags to designate packet groups. To control these bits, use the group statement, which is similar to an event range statement. See EVENT.DOC.

```
stream.pkt.<name>.group uid0(n) uid1(n)
```

The uid0(n) and uid1(n) arguments define a unit id range. You may specify more than one group by defining more than one group statement. The uid1(n) argument must be greater than uid0(n). Do not overlap ranges, and do not let groups intersect packet sections.

VARIABLE LENGTH PACKETS

There are two ways to specify variable length packets. In the first method, varLenEvent(ename) points to a value event, which associates unit ids with packet lengths. The second method specifies a list of lengths, which the stream repeats over and over. In either case, variableLength(1) must be set.

method 1.

```
stream.pkt.<name> varLenEvent(xx) ...
event.value.xx.range uid0(1) uid1(20) v(100)
event.value.xx default(50)
```

method 2.

```
stream.pkt.<name>.length L0(n) ... L9(n)
```

TIME SEGMENTS
To create packets out of chronological order, use time segment statements. The time segment statement is similar to group statements and to event range statements. The u0(n) and u1(n) arguments specify the starting and ending unit ids for a segment. A packet stream may have multiple time segment statements, and they may overlap.

```
stream.pkt.<name>.tseg u0(n) u1(n)
```

The unit ids refer to the packets as if they had been created in time order. For example,

```
stream.pkt.name.tseg u0(10) u1(15)
stream.pkt.name.tseg u0(10) u1(15)
stream.pkt.name.tseg u0(13) u1(1000)
```

The packet stream will create packets 10-15, then 10-15 a second time, and finally packets 13-1000. It then terminates. If a packet region is reading a raw file, you should set extendPastEOF(1) for that region. Otherwise, setting a time segment that goes beyond the file's EOF will cause the stream to shut down.

4.15 RECORD STREAM

This stream lets you design custom units through the script. You may define regions and fill them using any pattern, and you may use any task. A record stream will not read an input unit file, and it makes only fixed size units. The stream type must be "record." You may choose any name for <name> provided it is different from any other stream name. See STREAM.DOC for general stream information.

```
stream.record.<name> length(n) sizeBits(n) max(n:0) ERerrors(1or0:1) -
            ERdrop(1or0:1) drop(ename) encode(1or0:0)
stream.record.<name>.region.<rname> type(name) startbit(n) bits(n)
            startbyte(n) bytes(n)
```

Every record stream should have one or more region statements that tell it how to fill units. Unlike all other streams, you must define the region start and length in the region statement. The region names are insignificant except that two regions may not have the same name. Use the name for description. See REGION.DOC for syntax, but note that record regions demand additional location arguments.

The record stream may perform tasks. See TASK.DOC for sytax. It applies a task after filling regions but before error insertion. The task path name is "task."

```
stream.record.<name>.task.<task> event(ename) ...
```

Example:
100-byte units are constructed by three regions. "Header" and "trail" are fixed patterns. "Body" is constructed from an input stream.

```
stream.record.custom
stream.record.custom length(100)
stream.record.custom.region.Header type(F) pattern(0x1a) startbyte(0) bytes(8)
stream.record.custom.region.trail type(F) pattern(0x55) startbyte(92) bytes(8)
stream.record.custom.region.Body type(C) startbyte(8) bytes(84) inStream(pkt)
```

You can also use a record stream to pack variable length units into fixed size units. Simply define one region that encompasses the entire unit and use the C (consumer) pattern to fill it.

----

length(n)
Record size in bytes. It overrides sizeBits(n) if present. Either length(n) or sizeBits(n) is required.

sizeBits(n)
Record size in bits. length(n) will override it. Either length(n) or sizeBits(n) is required.

encode(1or0:0)
Set this argument to 1 if you plan to use the CRC or Reed-Solomon encoder task to encode the unit. Default=0. If you do not properly set this argument, then the record stream will not correctly handle the error convey() argument from input stream error statements.

```
stream.record.<name>.region.<rname> type(tname) startbit(n) bits(n)
                startbyte(n) bytes(n)
```

See REGION.DOC for more information.

type(tname)
Region type. See REGION.DOC.

startbyte(n)
Region start byte. It must be less than the unit byte length. Either startbyte(n) or startbit(n) is required. startbit(n) has precedence.

startbit(n)
Region start bit. It must be less than the unit bit length. Either startbyte(n) or startbit(n) is required. startbit(n) has precedence.

bytes(n)
Length of region in bytes. Either bytes(n) or bits(n) is required. bits(n) has precedence. 0= set to maximum length.

bits(n)
Length of region in bits. Either bytes(n) or bits(n) is required. bits(n) has precedence. 0= set to maximum length.

4.16 REGIONS

A region is an area within a unit, which is defined by start bit/byte and length, that a stream fills with one of several patterns. A pattern may be as simple as a fixed or ramp pattern, or it may involve file data or complicated custom applications. A region is identified by a region statement, which must be attached to a stream. Most streams require at least one region statement.

The Tiger core library provides eight patterns, and the CCSDS library provides two more. Applications may provide additional patterns. Here are the statements for the Tiger core patterns:

```
stream.<type>.<stream>.region.<region> checksum(1or0:0) type(U) -
      wrap(1or0:1) bits(n) format(c:D)
stream.<type>.<stream>.region.<region>.data s0(n) s1(n) s2(n) s3(n) ... s9(n)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(F) -
      pattern(0xnn:0xc9)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(A)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(E)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(S) start(n:0) -
      step(n:1) repeat(n:0)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(I) -
      inStream(sname) emptyUnit(c:F) fill(0xnn:0xc9) perfectFit(1or0:1)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(R) -
      device(dname) fill(0xnn:0xc9) extendPastEOF(1or0:0)

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(C) -
      inStream(sname) integral(n:0) integralStrategy(n:1) emptyUnit(c:F) -
      lastUnit(c:F) fill(0xnn:0xc9) ERcomposition(ename:0) -
      ERslip(1or0:1) ERerrors(1or0:1) startupOffset(n:0)
stream.<type>.<stream>.region.<region>.shortslip when(c) bits(n) event(ename)
stream.<type>.<stream>.region.<region>.longslip when(c) bits(n) event(ename)
```

The <stype> and <stream> paths are respectively the type and name of the stream to which a region is attached. The <region> field is the region name. Each stream type specifies the regions by name that you must define in the script. For example, many streams require that you define a "data" region. For those streams, you must include a region statement with "data" substituting for

"<region>." See each stream's documentation for region names. You do not define the start bit and length of a region because each stream automatically determines them. (The only exception is the record stream, which expects you to define a region's location in a unit.)

type(rname)
Region type. It describes how the region is filled. There is usually no default, but some streams may demand a particular pattern, in which case this argument is omitted. Each type, except for type A (random pattern) and type E (empty), has additional arguments, which are described below. Special applications may define additional types.

Name	Description
A	random number sequence
U	user defined sequence
F	fixed pattern
S	step pattern
R	raw file
C	input units
I	inlay pattern
E	empty region. No fill

The following patterns are CCSDS patterns and are usually used in secondary headers. See the TIME.DOC.
  tcday   day segmented time
  tccuc   unsegmented time

checksum(1or0:0)
1= The stream substitutes a checksum (sum of all previous region bytes) in the last byte of the region. default=0. The region must be byte-aligned and greater than two bytes long for this option to be allowed. You should not use a checksum in certain stream regions, such as CADU or V1TF data regions, because it violates CCSDS specifications.

The rest of this section defines region arguments by region type.

----
type U (user defined pattern)
You define a string of bytes in the script to be inserted in a region. If the user pattern is shorter than the region, the stream will repeat the pattern to fill it. You may enter the pattern in decimal, hex, or as text.

```
stream.<type>.<stream>.region.<region> checksum(1or0:0) type(U) -
      wrap(1or0:1) bits(n) format(c:D)
stream.<type>.<stream>.region.<region>.data s0(n) s1(n) s2(n) s3(n) ... s9(n)
```

wrap(1or0:1)
0= restart user pattern for each region.
1= begin next region with leftover bytes from current region. default=1.

bits(n)
Number of bits in the user pattern. 0= derive length from user pattern.

format(c:D)
Input format. D=decimal bytes. H=hex bytes. S=text. Default=D.

When using a user string, you must define one or more of the following lines, which define the user pattern. Each sx() argument defines one byte (decimal or hex) or a string (string format). Not all ten values must be defined on each line.

stream.<type>.<stream>.region.<region>.data s0(n) s1(n) s2(n) s3(n) ... s9(n)

examples:
stream.pkt.ap1.region.data.data s0(100) s1(50) s2(23)
stream.pkt.ap1.region.data.data s0(0xff) s1(0x12) s2(0xee)
stream.pkt.ap1.region.data.data s0(The end is near.)

----
type F (fixed pattern)
The region pattern is a constant byte value. example: c9c9c9c9c9c9c9c9

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(F) -
pattern(0xnn:0xc9)

pattern(0xnn:0xc9)
This is the byte pattern in hexadecimal. default=0xc9.

----
type S (step pattern)
The region is filled with a step/ramp pattern. example: 1111222233334444

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(S) start(n:0) -
step(n:1) repeat(n:0)

start(n:0)
Starting byte value in decimal. default=0.

step(n:1)
Step increment in decimal. default=1.

repeat(n:0)
Number of times to repeat the start byte before incrementing by the step value.
default=0, which increments after each byte.

----
type I (inlay pattern)
The region is filled with exactly one input unit. The unit must fit the region; it cannot be too big.

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(I) -
inStream(sname) emptyUnit(c:F) fill(0xnn:0xc9) perfectFit(1or0:1)

inStream(sname)
Input stream name. Required. The input stream must be defined in the script.

emptyUnit(c:F)
How the pattern fills an empty region if there are no input units but there are regions to fill. F=fixed pattern fill, N=no fill (leave as-is), I=idle units, X=exit, which causes the stream to shut down. Default=F.

fill(0xnn:0xc9)
Fill byte if the input unit is smaller than the region. Also used when emptyUnit=F. default=0xc9.

perfectFit(1or0:1)
1= the input unit must exactly fit the region. It may not be too big or too small. Default=1.

----
type R (raw file pattern)

The stream fills the region by reading bytes from a file. It reads as many bytes as it needs to fill a region and continues from where it left off with each new unit region. It uses a constant fill byte for the last region if the file is depleted.

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(R) -
device(dname) fill(0xnn:0xc9) extendPastEOF(1or0:0)

device(dname)
Device name of device from which it will get bytes. See DEVICE.DOC. The target device statement must be defined in the script. Required.

fill(0xnn:0xc9)
Constant fill byte when file is depleted. default=0xc9.

extendPastEOF(1or0:0)
1= continue filling regions with fill after file is depleted. Default=0.
0= stop stream. If the loader is position past the end of file marker, the stream will shut down unless extendPastEOF=1.

----
type C (consumed units)
The stream fills the region by loading unit from an input stream.

stream.<type>.<stream>.region.<region> checksum(1or0:0) type(C) -
inStream(sname) integral(n:0) integralStrategy(n:1) emptyUnit(c:F) -
lastUnit(c:F) fill(0xnn:0xc9) ERcomposition(ename:0) -
ERslip(1or0:1) ERerrors(1or0:1) startupOffset(n:0)
stream.<type>.<stream>.region.<region>.shortslip when(c) bits(n) event(ename)
stream.<type>.<stream>.region.<region>.longslip when(c) bits(n) event(ename)

inStream(sname)
Input stream name. Required. The input stream must be defined in the script.

integral(n:0)
0= Units are loaded back-to-back and split between regions if necessary.
>0 = The pattern loads an integral number of input units (n) into each region.
It adds fill (see lastUnit) to fill out a unit if necessary. It does not split an input unit unless integralStrategy=0 and the first input unit is bigger than the region. default=0.

integralStrategy(n:1)
What to do if the region is smaller than the first input unit.
0= split the input unit as if integral=0. Do not abort.
1= Abort. default=1.

emptyUnit(c:F)
How the pattern fills an empty region if there are no input units but there are regions to fill. This option pertains to a region that has not been loaded at all. If the region is partially filled, it used lastUnit(c) instead. default=F. F=fixed pattern fill, N=no fill (leave as-is), I=idle units, X=exit, which causes the stream to shut down.

lastUnit(c:F)
How the pattern fills the last, partially filled region when there are no more input units. It also determines how it fills partial regions when integral>0. default=F. F=fixed pattern fill, N=no fill (leave as-is), I=idle units. X is not an option for lastUnit().

fill(0xnn:0xc9)
Fill byte when lastUnit=F or emptyUnit=F. default=0xc9.

ERcomposition(ename:0)
When active, the stream writes a list of input units that are contained in each unit's region. It also notes the "first header pointer," which is an offset to the first input unit header (or unit start) and the number of bits or bytes of the last input unit. 1= write composition for every unit. 0= do not show composition. default=0. Otherwise, the value is a boolean event name. The stream prints composition only when the event is true. This lets you retrict output by specifying active ranges. See EVENT.DOC.

ERslip(1or0:1)
1= write slip information to the expected results file. 0= do not write slip information. default=1.

ERerrors(1or0:1)
1=identify regions that have inherited errors from input units to the expected results file. 0=ignore inherited errors. default=1.

startupOffset(n:0)
The pattern skips n bits of the first region before loading the first input unit. It zeroes the skipped bits. default=0.

The consumer pattern may slip bits as it loads units. It may slip short or long any number of bits. Short slips are trimmed from the end of input units. Long slips are zero bits at the beginning of input units. To slip short or long, add the additional script lines:

stream.<type>.<stream>.region.<region>.shortslip when(c) bits(n) event(ename)
stream.<type>.<stream>.region.<region>.longslip when(c) bits(n) event(ename)

when(c)
Determines when to slip. Required, no default. N=Never. Do not slip. F=First. Slip the first input unit per region only. A=All. Slip every input unit in a region.

bits(n)
Number of bits to slip. Required. minimum=1. WARNING! THE CURRENT TIGER VERSION ONLY SUPPORTS BYTE-WIDE SLIPS. You may only slip in multiples of eight bits.

event(ename)
The event defines which units (by unit id) will be slipped. See EVENT.DOC. The target event statement must exist in the script. If you set event(1), then the stream slip every unit.

4.17 STREAM

This is the general stream definition from which all other streams are derived. Not all arguments apply to all streams. Exceptions are noted below.

Many streams can be set up to get units from a file instead of making them. The key arguments to do this are fileType(c) and device(dname). All CCSDS streams allow this diversion. There is also a generic unit-reader stream called "depot," which is dedicated to reading units from a file. See DEPOT.DOC.

stream.<type>.<name> max(n:0) sizeBits(n) ERerrors(1or0:1) ERgaps(1or0:1) -
ERdrop(1or0:1) device(dname) fileType(c) stepSize(n:1) -
startSequence(n:0) skipZero(1or0:0) drop(ename)

max(n:0)
The stream will stop producing units when the unit id exceeds the specified id. default=0 (unlimited). Not required. If omitted, there is no limit. This argument may also be called maxUnitId(n).

sizeBits(n)
Unit size in bits. Some streams may not use this argument and instead may define a custom argument. Check each stream definition for details.

ERerrors(1or0:1)
1= put unit error information into the expected results file. default=1.

ERgaps(1or0:1)
1= put unit gap information in the expected results file. default=1. It is if the units do not have sequence numbers.

ERdrop(1or0:1)
1= identify discarded units in the expected results file. default=1.

device(dname)
Device name of file that contains input units. See DEVICE.DOC. The existence of this argument causes the stream to read units from a device instead of making them. This argument does not apply to all streams. See each stream definition for details. A device statement must be defined in the script.

fileType(c)
Device file type. P=plain unit file. F=fancy unit file. It is required if device(dname) is specified.

stepSize(n:1)
Sequence number step size if the stream creates units that have sequence numbers. It does not apply to all streams. It may be positive or negative but may not be zero. default=1.

startSequence(n:0)
Sequence number of the first unit if the stream creates units that have a sequence number. It does not apply to all streams. It may be positive or negative. default=0.

skipZero(1or0:0)
1= the stream skips sequence number zero if it creates units that have a sequence number. It does not apply to all streams. default=0.

drop(ename)
This argument causes the stream to discard units based upon the event. See EVENT.DOC. If omitted, the stream does not drop any units. Not required. If present, the event must exist in the script.

4.18 TASKS

A task is something that a stream or an output does to a unit. You define a task using a task statement, which you must attach to a stream or to an output. Tasks are linked to events. The event tells the stream or output when to apply the task. Tasks behave much like errors. In fact, an error is a specialized task. The primary use of a task is to insert values into units. The advantage of a task is that it allows the user to change the characteristics of a unit through the script without reprogramming the unit construction. For example, you can use tasks to deposit special values into units, which may be beyond the scope of a stream.

The following tasks are available:

set	deposit a 1-32 bit value anywhere in a unit.
add	add a 1-32 bit value to the existing 1-32 bit value anywhere in a unit.
flip	invert 1+ bits anywhere in a unit.
flipmask	invert 1-32 bits anywhere in a unit based upon a care/don't care mask.

resize	change a unit's size by truncation or extension. You can either designated a target length or you may specify the number of bit/bytes to extend or truncate a unit.
sequence	insert a sequence number anywhere in a unit. The sequence number automatically increments by the step value after each application.
pn	pseudo noise encode the unit, skipping any sync pattern. This is the CCSDS version of the encoder.
crc	crc encode the unit. The encoder puts the 16-bit CRC in the last two bytes of the unit.
rs	Reed-Solomon encode the unit. The encoder puts the parity in the last bytes of the unit. (paritySize = 32 * interleave.) The unit size must be (codeWordLength * interleave + syncLength) bytes long.

Support for tasks is stream and output dependent. See the documentation for the stream or output component for more information. The general formats are:

stream.<stype>.<stream>.<task>.flip event(ename) startbit(n) bits(n)

stream.<stype>.<stream>.<task>.set event(ename) startbit(n) bits(n) v(n)

stream.<stype>.<stream>.<task>.add event(ename) startbit(n) bits(n) v(n)

stream.<stype>.<stream>.<task>.flipmask event(ename) startbit(n) bits(n) -
emask(0xnnnnnnnn)

stream.<stype>.<stream>.<task>.resize label(name) event(ename) -
chop(1or0) fill(0xnn:0) v(n) bits(1or0:0) abs(1or0:0)

stream.<stype>.<stream>.<task>.sequence event(ename) startbit(n) bits(n) -
stepSize(n:1) startSequence(n:0) skipZero(1or0:0)

stream.<stype>.<stream>.<task>.pn event(ename) skipBytes(n:0)

stream.<stype>.<stream>.<task>.crc event(ename) syncBytes(n:4) length(n) -
CRCstart(0xnnnn:0xffff) CRCincludeSync(1or0:0)

stream.<stype>.<stream>.<task>.rs event(ename) syncBytes(n:4) -
RSdual(1or0:1) RSinterleave(n:4) RScodeLength(n:255)

The <stype> and <stream> are the type and name of the stream to which the task is attached. For outputs, substitute the output prefix for stream.<stype>.<stream>. Set <task> to either "task" or "xtask." Use "task" to apply the task before the stream encodes units, and use "xtask" to apply the task after the stream encodes units. Some streams may not support both forms, and others may add additional <task> paths. See the appropriate stream documentation for details.

For tasks that duplicate errors, see ERROR.DOC. Note that these tasks do not use the convey(1or0) argument. These tasks include:

flip	invert a consecutive string of bits.
set	deposit a 1-32 bit value somewhere in a unit.
add	add a 1-32 bit value somewhere in a unit.
flipmask	flip up to 32 bits corresponding to "on" bits in a 32-bit mask.
resize	truncate or extend a unit

event(ename)
Event name. event(1)= always true; apply to every unit. Required. An event statement must be in the script. See EVENT.DOC.

skipBytes(n:0)
Number of bytes in the sync pattern. Default=0. The PN encoder skips this many bytes from the start of the unit.

stepSize(n:1)
Sequence number step size. It may be positive or negative but may not be zero. default=1.

startSequence(n:0)
Starting sequence number. It may be positive or negative. default=0.

skipZero(1or0:0)
1= the stream skips sequence number zero. default=0.

startbit(n)
The startbit and bits arguments define a location in a unit where the task is applied. Startbit may be positive or negative. When zero or positive, the stream measures the location from the start of the unit with the first bit being bit zero. The maximum value for startbit is the unit size in bits minus one. When startbit is negative, the stream measures the location from the end of the unit. For example, startbit(-1) points to the last bit, and startbit(-16) points to 16 bits from the unit end. Required.

bits(n)
The number of bits in the task field. See startbit(n) description. For flip, it may be as large as the unit size in bits minus startbit(n). If you set bits(0), the stream inverts all bits in the unit starting with startbit(n) to the end of the unit. For all other task types, the range is 1 to 32 bits. Required.

syncBytes(n:4)
Number of bytes in sync pattern. The task skips this many before performing the task regardless on whether the unit has a real sync pattern or not. Default=4. Range=0-4.

length(n)
Unit length in bytes. Required. min=4.

CRCstart(0xnnnn:0xffff)
The initial CRC16 encoding value. Default=0xffff.

CRCincludeSync(1or0:0)
1= include sync pattern in CRC encoding. Default=0. See syncLength(n).

RSdual(1or0:1)
1= assume dual mode. Default=1.

RSinterleave(n:4)
RS interleave. range=1-32. Default=4.

RScodeLength(n:255)
Reed-Solomon codeword length in bytes. Default=255. range=33-255.

4.19 TELECOMMAND PHYSICAL CHANNEL STREAM

This stream makes CLTUs with acquisition and idle sequences. It requires plain CLTUs as input. It may be used in the TC scenario only. See STREAM.DOC for general stream information.

The stream type must be "tcphy." You may choose any name for <name> provided it is different from any other stream name.

stream.tcphy.<name> inStream(sname) acquisition(n:128) start(n:1) idle(n:8) -

max(n:0) sizeBits(n:0) ERerrors(1or0:1) drop(ename) -
ERgaps(1or0:1) ERdrop(1or0:1) device(dlabel) fileType(c)

This stream may perform tasks. See TASKS. It applies "task" to units before error insertion and "xtask" to units after error insertion.

stream.tcphy.<name>.task.<name> event(ename) ...
stream.tcphy.<name>.xtask.<name> event(ename) ...

inStream(sname)
Input stream name. It should be a stream that provides plain CLTUs. Required.

acquisition(n:128)
Number of acquisition sequence bits. Default=128. This number must be byte divisible. This argument is ignored if acqEvent(ename) is specified.

acqEvent(ename)
Name of value event that specifies the acquisition sequence length in bits for the CLTUs. All event values must be byte divisible. If specified, acquisition(n) is ignored. See EVENT.DOC.

start(n:1)
acquisition start bit value, 1 or 0. Default=1.

idle(n:8)
Number of idle sequence bits. Default=8. This number must be byte divisible.

sizeBits(n:0)
If greater than zero, it defines the bit length of all output CLTUs. All output CLTUs will have the same size defined by sizeBits(n). The stream will extend the idle sequence if necessary. If an input CLTU is bigger than sizeBits(n), then this stream aborts. Default=0.

See STREAM.DOC for undefined arguments.

4.20 TELECOMMAND TRANSFER FRAME

This stream makes Telecommand Transfer Frames (TCTFs) from either source packets or telecommand segments. It may only be used in the TC scenario. See STREAM.DOC for general stream information.

This stream can read TCTFs from a file instead of making them. You may also use the depot stream to read TCTFs from a file. See DEPOT.DOC.

The stream type must be "tctf." You may choose any name for <name> provided it is different from any other stream name.

stream.tctf.<name> inStream(sname) spid(n) vcid(n) length(n:1024) -
crc(1or0:0) CRCstart(0xnnnn:0xffff) segment(1or0:0) -
aggregate(1or0:1) version(n:0) control(ename) max(n:0) -
ERerrors(1or0:1) ERgaps(1or0:1) ERdrop(1or0:1) drop(ename) -
device(dname) fileType(c) stepSize(n:1) startSequence(n:0) -
skipZero(1or0:0)

This stream may perform tasks. See TASKS. It applies "task" before error insertion and encoding, and "xtask" after error insertion and encoding.

stream.tctf.<name>.task.<task> event(ename) ...
stream.tctf.<name>.xtask.<task> event(ename) ...

----
inStream(sname)
Input stream name. It should be a stream that provides either source packets or telecommand segments. Required unless reading from a file.

spid(n)
Spacecraft id. Required. Range=0-1023.

vcid(n)
Virtual channel. Required. Range=0-63.

length(n:1024)
Maximum frame length in bytes. Default=1024. Range=8-1024. Each TCTF length may be smaller depending on the size of the input unit. The input unit length (packet or tcsegment) must be no greater than length-5 if crc(0) or length-7 if crc(1).

crc(1or0:0)
1= encode frame with CRC16. Default=0.

CRCstart(0xnnnn:0xffff)
CRC start value if crc(1). Default=0xffff.

segment(1or0:0)
1= input stream is a TC segment stream or a mux connected to a TC segment stream. Default=0.

aggregate(1or0:1)
1= collect one more packet per TCTF. Meaningful only if segment(0). Default=1.

version(n:0)
Frame version. Default=0. Range=0-3.

control(ename)
This event must be a value event. It identifies the BYPASS and CONTROL COMMAND bit field for each output frame. Value definition: 0= type-AD, 2= type-BD, 3=type-BC. If control(ename) is omitted, the default is 0=type-AD. See EVENT.DOC.

See STREAM.DOC for undefined arguments.

4.21 TIME

This section describes NASA and CCSDS timecode formats, which Tiger treats as unit regions. See REGION.DOC for region information. They automatically increment by the step value whenever they are loaded into a unit.

Timecode patterns are usually loaded into packet or frame secondary headers. When you use a timecode as a secondary header, you must set the parent stream's 2hdrLength() argument to the proper length, which must be greater than or equal to the timecode length. If you choose a secondary header length that is larger than the timecode length, then the stream will insert zeroes in the extra bytes.

Tiger supports the following formats. See the appropriate document.

type	description	document
tcuc	unsegmented	TCUC.DOC
tcday	day segmented	TCDAY.DOC
pb5ja	PB-5J	PB5J.DOC
pb5jb	PB-5J	PB5J.DOC
pb5jc	PB-5J	PB5J.DOC
pb5jd	PB-5J	PB5J.DOC

4.22 EDU STREAM

The ETS EDU stream makes EDOS packet SDUs. A packet SDU consists of an ESH (EDOS Service Header) and a pakcet. The stream should be attached to a single packet stream via inStream(sname). See PACKET.DOC for packet stream information.

The stream type must be "edu." You may choose any name for <name> provided it is different from any other stream name.

The ESH contains a field that identifies a packet length error. This stream does not use a script argument to control that flag. Instead, it sets that flag by comparing the expected packet length to the length it fetches from the packet data. If you want to create packet length errors for selected SDUs, then introduce bit errors to the packet length field in the input packet stream. You cannot generate packet length errors directly in the edu stream except with the clever use of tasks.

The ESH contains a field that identifies packet sequence errors. This stream does not use a script argument to control that flag. Instead, it sets it by tracking the sequence counter, which it fetches from the packets. If you want to create packet sequence errors, then you must introduce them in the input packet stream by either dropping packets or by putting bit errors in the packet sequence field. As with packet length errors, you cannot generate sequence errors directly in the edu stream. The sequence checker expects the packet sequence increment to be +1 and will generate sequence errors for any other step size.

When a VCDU sequence error occurs, the first packet from the next VCDU should be marked as having a packet sequence error. The stream does NOT automatically do this because it does not know which packet is the first. You will need to insert the proper script statements to achieve this effect.

If you connect more than one packet stream to a single edu stream, then you should expect to see many packet sequence errors. To properly handle multiple packet streams, create an edu stream for each packet stream, and then mux the edu streams.

The edu stream does not make SDUs from idle packets. It discards idle packets.

The pb5 support statement is required because it defines the timekeeping mechanism that the stream uses to stamp every SDU. See PB5.DOC for syntax information.

```
stream.edu.<name> inStream(sname) version(n:0) port(n:0) playback(1or0:0) -
        capture(1or0:0) spid(n:42) vcid(n:0) vcduBreak(ename) -
        RSuncorr(ename) RSchs(ename) RScs(ename) fillEvent(ename) -
        fill(0xnn:0x00) max(n:0) drop(event) ERerrors(1or0:1) -
       ERdrop(1or0:1)
stream.edu.<name>.pb5 day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0) -
        dayStep(n:0) secondsStep(n:1) milliStep(n:0) microStep(n:0) -
        ramp(n:0) drift(n:0) driftFreq(n)
```

This stream may perform tasks. See TASKS. It applies "task" to units before error insertion and "xtask" to units after error insertion.

stream.edu.<name>.task.<task> event(ename) ...
stream.edu.<name>.xtask.<task> event(ename) ...

----
inStream(sname)
Input stream name. This should be a stream that provides packets. Required unless reading from a file.

version(n:0)
ESH version number. Default=0. Range= 0-15.

port(n:0)
ESH port number. Default=0. Range= 0-63.

playback(1or0:0)
ESH playback flag. Default=0.

capture(1or0:0)
ESH recovery process indicator. 1=data capture playback. 0=live. Default=0.

spid(n:42)
Spacecraft id. Default=42.

vcid(n:0)
Virtual channel id. Default=0.

vcduBreak(ename)
Boolean event that identifies which SDUs should have the "source VCDU sequence counter discontinuity" flag set. See EVENT.DOC. If omitted, no SDUs have the flag set. This may not be meaningful for packet SDUs.

RSuncorr(ename)
Boolean event that identifies which SDUs should have the "Reed-Solomon control flag" set. See EVENT.DOC. If omitted, no SDUs have the flag set. This may not be meaningful for packet SDUs.

RSchs(ename)
Value event that identifies which SDUs have Reed-Solomon header corrected symbols. The value is the number of corrected header symbols in the parent VCDUs. See EVENT.DOC. If omitted, all SDUs have zero Reed-Solomon header corrected symbols. This should not be meaningful for EDOS because the CADU headers are not encoded.

RScs(ename)
Value event that identifies which SDUs have Reed-Solomon corrected symbols. The value is the number of corrected symbols in the parent VCDUs. See EVENT.DOC. If omitted, all SDUs have zero Reed-Solomon corrected symbols.

fillEvent(ename)
Value event that identifies which SDUs contain short packets. The value is the number of fill bytes at the end of a packet. The stream will overwrite the requisite number of trailing bytes in affected packets with the fill pattern. If omitted, there are no short packets.

fill(0xnn:0x00)
The fill byte for short packets. Default=0x0.

See STREAM.DOC for undefined arguments.

4.23 ESTP PRODUCTS

The "etsp" program creates ETS data products. These include:

a. **Production Data Sets (PDS)**
b. **Expedited Data Sets (EDS)**
c. **Rate Buffered Packet files containing packet EDUs**
d. **EDU files**

To run the program, type:

etsp <script> [<erfile>]

Unlike sctgen and tccsds, the script does not require a "main" statement. Each script may create only one output product.

If you omit arguments, etsp prints its version number and a syntax message. The second file, the expected results file name, is optional. If omitted, etsp derives the name from the script file name.

A PDS or EDS consists of a construction record file plus one or more data files. The Rate Buffered Packet file (RBP) is a file of packet EDUs from one application id. The EDU file is nearly identical to the RBP file except for two differences. First, the RBP file contains only one appid while the EDU file may contain packet SDUs from multiple appids. Second, the RBP file has a precise naming convention while the EDU file does not.

To understand the contents of these products, see the following ICD:

EOS Data and Operations System (EDOS), CDRL B301, August 9, 1996.

For information about building scripts for the data products, see the following documents:

PDS.DOC
EDS.DOC
RBP.DOC
EDU.DOC

4.24 PDS AND EDS

This section describes how to make a Production Data Set (PDS) and an Expedited Data Set (EDS), which may consist of multiple files. You should have your EOS ICD open to table 8.1.2.7-1, PDS/EDS Construction Record, page 8-11, as you make the script because most of the dataset output arguments directly affect the construction record fields.

The PDS/EDS script contains the following components:
one PDS/EDS output (see below)
one mux if there are multiple packet streams (see MUX.DOC)
one or more packet streams (see PACKET.DOC)

Each packet stream produces packets for one appid. The PDS/EDS output assumes all packet streams are from the same virtual channel.

If you wish to introduce gaps to a packet stream, either use the drop(ename) argument in the packet stream statement, or insert bit errors in the packet sequence field.

If you wish to introduce packet length errors, either insert bit errors in the

packet length field in the packet stream or make packets that do not match the lengths specified in the output.

See PACKET.DOC in the Tiger reference documents for the packet statement syntax.

PDS/EDS OUTPUT

The PDS/EDS output engine creates the construction record file and the packet data files. To create an EDS, use "output.eds." To create a PDS, use "output.pds." The supporting output.eds.esh or output.pds.esh statement is required. See PB5.DOC. The SCSstart and STSstop statements are optional and define start and stop time segments, which it writes to the construction record. One output.eds.appid or output.pds.appid statement is required for each appid. It defines parameters for the appid, which must match a packet stream appid.

```
output.pds inStream(sname) fillEvent(ename) fill(0xnn:0xc9) -
      KBperFile(n:0) max(n:0) major(n:0) minor(n:0) spid(n:42) -
      create(n:97100050403) dscount(n:0)

output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
      length(n) length1(n) length2(n) length3(n) discardBadLenPkt(1or0:1)
output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
      length(n) length1(n) length2(n) length3(n) discardBadLenPkt(1or0:1)
output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
      length(n) length1(n) length2(n) length3(n) discardBadLenPkt(1or0:1)

output.pds.esh day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0) -
         dayStep(n:0) secondsStep(n:1) milliStep(n:0) microStep(n:0) -
         ramp(n:0) drift(n:0) driftFreq(n)

output.pds.SCSstart day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0)
output.pds.SCSstop day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0)

---------------------------

output.eds inStream(sname) quicklook(1or0) fillEvent(ename) fill(0xnn:0xc9) -
      KBperFile(n:0) max(n:0) major(n:0) minor(n:0) spid(n:42) -
      create(n:97100050403) dscount(n:0)

output.eds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
      length(n) length1(n) length2(n) length3(n) discardBadLenPkt(1or0:1)
output.eds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
      length(n) length1(n) length2(n) length3(n) discardBadLenPkt(1or0:1)
output.eds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
      length(n) length1(n) length2(n) length3(n) discardBadLenPkt(1or0:1)

output.eds.esh day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0) -
         dayStep(n:0) secondsStep(n:1) milliStep(n:0) microStep(n:0) -
         ramp(n:0) drift(n:0) driftFreq(n)

output.eds.SCSstart day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0)
output.eds.SCSstop day(n:0) seconds(n:0) milliseconds(n:0) microseconds(n:0)
```

----

quicklook(1or0)
Quicklook construction method. Required for EDS. The PDS output ignores it.

1= Make EDS based on quicklook flag in packet secondary header. The output discards packets that do not have the quicklook flag set.
0= Make EDS from all packets, and ignore the quicklook flag.

inStream(sname)
Input stream name. Required. You must include a packet stream definition by that name in the script. See PACKET.DOC. The input stream must provide packets from one application id to satisfy the PDS/EDS requirements.

fillEvent(ename)
The construction record identifies short packets, which are packets that could not be completely constructed and are missing trailing bytes. The packet assembly system appends fill to short packets. This event points to a value event. When the event is true, the output chops the corresponding packet to a shorter length and replaces the lost data with fill. The occurrence is noted in the construction record. The event value is the number of chopped bytes. If fillEvent(ename) is omitted, then the PDS/EDS does not have any short packets. See EVENT.DOC for event syntax.

fill(0xnn:0xc9)
This argument identifies the fill byte when fillEvent(ename) is used. The default fill is 0xc9.

KBperFile(n:0)
A PDS/EDS consists of a construction record and one or more data files. This argument defines how many kilobytes should be written to each data file. The default is zero, which means the output writes all packets to a single data file. Note that a data file may be slightly larger or smaller than expected because the output engine does not split packets between files.

max(n:0)

Maximum number of packets that the output writes to the PDS/EDS before terminating the scenario. The default is zero, which is unlimited. You must set the max(n) argument in either the output or all packet streams to ensure that the program stops.

major(n:0)
The construction record major version number. Default=0.

minor(n:0)
The construction record minor version number. Default=0.

spid(n:42)
Spacecraft id. Default=42.

create(n:97100050403)
The PDS/EDS creation date, which is part of the PDS/EDS name. The format is "yydddhhmmss." The default is day 100 of 1997, 5 hours, 4 minutes, and three seconds.

dscount(n:0)
Data set counter, which is part of the PDS/EDS name. (It is the third from last character.) Range= 0-9. Default=0.

----

output.pds.esh . . .

output.eds.esh . . .

Required. See PB5.DOC
----
output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)
output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)
output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)

output.eds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)
output.eds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)
output.eds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)

Each appid statement defines one PDS/EDS application id. You must have at least one and no more than three. Each one must match a corresponding packet stream appid.

appid(n)
Packet application id, which is written to the construction record. It must match the application id from an input packet stream. Required. rscPackets(n:0)

The construction record contains a field that counts the number of packets from Reed Solomon correctable frames. This argument is that number. The default is zero.

vcid(n:0)
Virtual channel number. Default=3D0. Range=3D 0-62.

vcid2(n) Second virtual channel number. If omitted, then there is only one virtual channel for this application id. Range=3D 0-62.

length(n)
length1(n)
length2(n)
length3(n)

Packet lengths in bytes. You may specify up to four acceptable packet lengths for each application id. Range: 7-65,542 bytes. Packets with unacceptable lengths are identified as bad length packets in the construction record. By default the output discards them. If you omit all length arguments then, with one exception, all packet lengths within 7-65,542 bytes are acceptable. The exception is the output will discard any packet whose internal packet length field does not match the actual length as provided by the packet stream. It will do this regardless of the value or existence of the length arguments.

Note: PDS Scripts created with previous versions of SCTGEN are not compatible with SCTGEN version 1.2. The following line must be added for each apid to run previously created scripts.

output.pds.appid id(n) rscPackets(n:0) vcid(n:0) vcid2(n) -
length(n) length1(n) length2(n) length3(n)

----

output.pds.SCSstart day(n:0) seconds(n:0) milliseconds(n:0)

microseconds(n:0)
output.pds.SCSstop day(n:0) seconds(n:0) milliseconds(n:0)
microseconds(n:0)

output.eds.SCSstart day(n:0) seconds(n:0) milliseconds(n:0)
microseconds(n:0)
output.eds.SCSstop day(n:0) seconds(n:0) milliseconds(n:0)
microseconds(n:0)

These PDS/EDS output statements define the spacecraft start and stop times, which are written to the construction record. You should provide at least one pair. You may specify more than one start and stop time pair. The output does not check to see if the stop time is greater or less than the start time.

day(n:0)
Day from epoch. Range=3D 0-9999. Default=3D0.

seconds(n:0)
Range=3D 0-86399. Default=3D0.

milliseconds(n:0)
Range=3D 0-999. Default=3D0.

microseconds(n:0)
Range=3D 0-999. Default=3D0.

This example script makes an EDS with packet appids 1 and 2. Notice that the EDS output connects to two packet streams. This script creates 250 packets. It puts no more than 10 kilobytes in one file. It generates all of the possible error conditions, which are found in the construction record=. The EDS is not constructed on the packet secondary header quicklook flag. (To do that, set quicklook(1) in output.eds. Then in each packet stream, set quicklook(ename), and specify a unit event to identify which packets have the quicklook flag set. The second packet stream shows how to do that. In this scenario, we are putting all packets in the EDS, and we are ignoring the packet quicklook flag.)

This example etsp script makes a multi-file Expedited Data Set (EDS). It contains two appids. It is not triggered by the quicklook flag.

```
#_________________
output.eds inStream(merge) major(1) minor(2) KBperFile(10) quicklook(0)
        spid(42) create(97101050403) dscount(1) fillEvent(fill)
fill(0x0) -
       max(250)

output.eds.esh day(100) seconds(0) milliseconds(0) microseconds(0) -
        dayStep(0) secondsStep(0) milliStep(100) microStep(50)

output.eds.appid id(1) rscPackets(25) vcid(1) vcid2(2)
output.eds.appid id(2) rscPackets(0) vcid(0)

output.eds.SCSstart day(100) seconds(500) milliseconds(50) microseconds(0)
output.eds.SCSstop  day(100) seconds(1000) milliseconds(50) microseconds(0)
output.eds.SCSstart day(200) seconds(500) milliseconds(50) microseconds(0)
output.eds.SCSstop  day(200) seconds(1000) milliseconds(50) microseconds(0)

event.value.fill.range uid0(1) uid1(4) v(16)
```

```
#________________
stream.mux.merge default(p1)
stream.mux.merge.recur start(1) skip(1) stream(p2)

#________________
stream.pkt.p1 appid(1) length(100) 2hdr(1) 2hdrLength(9) drop(drop)
stream.pkt.p1.region.data type(F) pattern(0x12)
stream.pkt.p1.region.2hdr type(tcEDOS)
stream.pkt.p1.error.flip label(lengthErr) event(lenErr) startbit(32)
bits(16)
event.unit.drop.range uid0(25) uid1(26)
event.unit.lenErr.range uid0(50) uid1(54)

#________________
stream.pkt.p2 appid(2) length(200) 2hdr(1) 2hdrLength(9)
stream.pkt.p2.region.data type(F) pattern(0x22)
stream.pkt.p2.region.2hdr type(tcEDOS)
stream.pkt.p2.region.2hdr type(tcEDOS) day(100) msOfDay(0) microOfMs(0)=
 -
            msStep(1000) microStep(0) quicklook(qlevent)
event.unit.qlevent.recur start(1) skip(2)
```

This example script creates a multiple apid PDS from an external file. It is not triggered by the quicklook flag. It contains 3 apids. This example uses the depot stream, which is discussed in Section 4.7 of this document. This example contains 2 SCSstart/stop time combinations. The fileType must be specified as VPKT indicating variable packet lengths and the external filename used to create the PDS is pkt.dat.

```
#______________________

main c(pds)

output.pds  inStream(p1) major(1) minor(2) Kbperfile(16) quicklook(0) -=

         spid(42) create(97101050403) dscount(0) fillEvent(fill)
fill(0xC9) -
        max(300)

output.pds.esh  day(100) seconds(0) milliseconds(0) microseconds(0)  -
           dayStep(0) secondsStep(0) milliStep(0)
microStep(50)

output.pds.appid  id(0) vcid(1) length(200)
output.pds.appid  id(1) vcid(1) length(300)
output.pds.appid  id(2) vcid(1) length(400)

output.pds.SCSstart  day(100) seconds(500) milliseconds(50) microseconds(0)
output.pds.SCSstop  day(100) seconds(1000) milliseconds(50) microseconds(0)
output.pds.SCSstart  day(200) seconds(500) milliseconds(50) microseconds(0)
output.pds.SCSstop  day(200) seconds(1000) milliseconds(50) microseconds(0)

event.value.fill.range uid0(2) uid1(3) v(16)

#_____________________
```

stream.depot.p1 fileType(VPKT) device(d)
device.file.d name(pkt.dat) access(r)

4.25 SONY TAPE DEVICE

The SONY tape device allows your script to read or write directly to a SONY tape device. You may use the SONY device statement anywhere that a device statement is permitted. However, you may specify only one SONY device per script. Certain output products do not specify a device, so you must create them on disk and later copy them to tape. Make sure the tape unit is ready before you run the script. See DEVICE.DOC for device information.

device.sony.<name> name(file) access(c) tapeRecSize(n:1024) name(file) Name of file, which is written to the tape directory. Required. access(c) Access mode. r=read, w=write. Required, tapeRecSize(n:1024) The tape record size. This is not the unit size. Default=1024.

device.sony.<name> name(file) access(c) tapeRecSize(n:1024)

name(file)
Name of file, which is written to the tape directory. Required.

access(c)
Access mode. r=read, w=write. Required,

tapeRecSize(n:1024)
The tape record size. This is not the unit size. Default=1024.

4.26 SLIP STREAM

Slips units (usually frames, CADUs, or V1TFs) long or short.

```
stream.slip.<name> -
   inStream(sname) long(ename) short(ename) -
   longFill(n:0x0) shrink(1or0:0) fill(n:0x0) -
   er(1or0:1) length(n:1024)
```

----
inStream(sname)
Input stream name. Required.

short(ename)
Event name of a value event that identifies which units by unit number are to be slipped short. The event's values are the number of slip bits, which must be between 1 and 32 bits. If short(ename) is omitted, no units are slipped short, however you must specify either short(ename) or long(ename). The stream will not slip the first input unit. See EVENT.DOC.

long(ename)
Event name of a value event that identifies which units by unit number are to be slipped long. The event's values are the number of slip bits, which must be between 1 and 32 bits. If long(ename) is omitted, no units are slipped long, however you must specify either short(ename) or long(ename). If you mark the same input to be slipped both long and short, then it slips the unit long and ignores the short slip. See EVENT.DOC.

length(n:1024)
Block size. Min=512. The stream fills fixed size block with the slipped input units. You have little reason to use other than the default.

shrink(1or0:0)

The last output block is often not completely filled. 1=trim the last block to the smallest byte length, filling unused bits with zeroes. 0= fill unused bits and bytes with the fill pattern and do not shrink. Default=0.

fill(n:0x0)
Byte used as fill when shrink(0). The stream uses the least significant bits of fill to fill less than a byte. Default=0.

longFill(n:0x0)
Pattern used when units are slipped long to fill the vacated hole. The stream uses the least significant bits of longFill to fill less than a byte. Default=0.

er(1or0:1)
1=write message to expected results file to mark every slipped unit. 0=do not write slip events to the expected results file. Default=1.

The slip stream slips units 1-32 bits, long or short, to test frame synchronization hardware and software. The usual slip range is one to three bits. Note that bit slip is a time consuming job for the slip stream.

The unit counter for slip identification is the slip stream's own input counter, and it may not be the same as the unit id that the input streams create, especially if the input stream is a mux.

The usual practice is to connect the slip stream directly to an output. See OUTPUT.DOC. Then connect the slip stream to a frame stream or frame mux, such as CADUs or V1TFs. The following script fragment is an example.

```
output.plain inStream(main) max(1000)

stream.slip.main inStream(vcmux) long(evL) short(evS)
event.value.evL.recur start(10) skip(9) v(1)
event.value.EvS.range uid0(45) uid1(46) v(2)
event.value.EvS.range uid0(95) uid1(96) v(7)

stream.mux.vcmux.recur stream(vc0) start(1) span(2)
stream.mux.vcmux.recur stream(vc1) start(2) span(2)
```

The mux connects to some unidentified streams named vc0 and vc1. The slip stream slips every tenth unit one bit long. It slips units 45 and 46 two bits short, and unit 95 and 96 seven bits short.

4.27 RATE BUFFERED FILE

```
output.rbp spid(n:42) appid(n) ground(name:WSG) time(n:97001010203) -
        KBperFile(n:0) max(n:0) inStream(sname)
```

appid(n)
Required.

inStream(sname)
Required.

spid(n:42)

ground(name:WSG)
Required 3 characters.

```
time(n:97001010203)
Required 11 characters.

KBperFile(n:0)

max(n:0)
# This example etsp script makes a Rate Buffered Packet file of 100 SDUs.

output.rbp inStream(r) ground(WSG) KBperFile(0) appid(320) spid(42) -
       time(97001050403) max(50)

#__________________
stream.edu.r inStream(p) version(0) port(1) playback(0) -
          capture(0) spid(42) vcid(0) vcduBreak(vcbrk) -
           RSuncorr(vcbrk) RScs(evtr) fillEvent(fill) fill(0x00)
stream.edu.r.pb5 day(1) dayStep(0) seconds(2) milliseconds(3) -
          microseconds(4) secondsStep(1) milliStep(2) microStep(3)

event.value.fill.range uid0(3) uid1(3) v(180)
event.unit.vcbrk.range uid0(2) uid1(2)
event.value.evtr.range uid0(4) uid1(4) v(10)

#__________________
stream.pkt.p appid(320) length(780) 2hdr(1) 2hdrLength(9) checksum(1) -
          drop(drop)
stream.pkt.p.region.data type(U) wrap(0) format(S)
stream.pkt.p.region.data.data s0(appid=320 vcid=41 spid=42 len=780. )

stream.pkt.p.region.2hdr type(tcEDOS) day(200) msOfDay(0) microOfMs(0) -
        msStep(100) microStep(0) ramp(0) drift(1) driftFreq(97)

stream.pkt.p.error.flip label(lengthErr) event(lenErr) startbit(32) bits(16)
event.unit.lenErr.range uid0(44) uid1(44)

event.unit.drop.range uid0(45) uid1(46)
event.unit.drop.range uid0(100) uid1(100)
```

SECTION 5
USER DETAILED SCRIPTING TUTORIAL

This section contains a detailed sequence of steps to suide the user through the scripting process in creating data scenarios for generation. The GUI exemplifies the simple data generation and more complex and non-ETS specific issues may be addressed by reading this section. This section is made up of four sub-sections.

5.1 TIGER TUTORIAL: SIMPLE SCRIPT

Tiger is a collection of tools and libraries to make files of CCSDS units (packets, CADUs, V1TFs, CLTUs) to be used to test telemetry or spacecraft command processing equipment. It does not transmit test data to target equipment; you will need other software or hardware to do that. It only creates test data files.

Tiger is actually a framework for creating any type of unit. CCSDS is only one application. Programmers can use the Tiger components to make their own applications. (This feature is unavailable at this time.) This document focuses on the CCSDS application, which is called "tccsds," and it does not present the programming interfaces.

This document explains how to write scripts to make CCSDS units. It does not explain how CCSDS works, but you must understand the CCSDS architecture to effectively use tccsds. You will need to read the CCSDS blue books to get that background. You can find the blue books on the World Wide Web at

http://joy.gsfc.nasa.gov/CCSDS-DocLib.html

The Tiger CCSDS program is called "tccsds," and there are SunOS/solaris and IBM PC versions. (We will have an HP version later.) To run tccsds, type at the UNIX or DOS prompt:

tccsds <script_file_name>

If you type tccsds by itself, it will print the command syntax. Tccsds is a batch-like program, which does not require any special graphics capabilities. It reads the script and produces an output test data file and an expected results file. Sometimes a script will reference other input files. See caduSample.script and caduSample.er for commented examples of each file type.

The script and the expected results file are text files that you can print or edit using any standard text editor. The script file contains tccsds setup information. The expected results file is a signature of what is in the corresponding binary test data file. We are currently developing GUI programs that handle both files. The script GUI will let users create and edit scripts. The expected results GUI will let users see the contents of a test data file in a friendlier manner.

Here is an example of a very user0 script. In this scenario, tccsds will make 1,000 packets and will write them to a test data file. The file will be binary, and the packets will be back-to-back with no special headers or trailers. (The dash lines are not part of the script. We will use them in this document to outline a script listing.) All script file names should have the extension "script" or "scr."

```
-------------
# user0.script
# A very user0 tccsds script that creates 1000 packets and writes them to
# a test data file.

main c(cadu)

output.plain device(user0) inStream(Packets) max(0)
device.file.user0 name(user00.dat) access(w)

stream.pkt.Packets max(1000) appid(10) tc(0) length(50) 2hdr(0) -
```

```
   checksum(0)
stream.pkt.Packets.region.data type(F) pattern(0x10)

*end
-------------
```

The "#" character in column one identifies a comment line. Tccsds ignores lines that begin with this character. It also skips blank lines. Comments may not appear on lines that contain setup information.

The "*end" line marks the last line in a script. Tccsds will not process lines that follow it. "*end" is optional, and we will omit it from all subsequent examples.

Script lines may be up to 512 characters in length. The "-" character, when it is the last character in a line, is a continuation character. Tccsds will join the primary line and all continuation lines into one logical line, which we call a STATEMENT. This lets you split long statements so that they are more readable, and it also lets you make statements that would exceed the 512 character limit. Continuation lines must follow the primary line with no intervening statements, but intervening comment lines are ok.

A statement consists of two parts: a path and an argument list, which must appear in that order. White space separates the two parts, and it also separates arguments. Do not put white space where it is unexpected because it will confuse the parser and cause errors.

Case is always important. You must exactly match the case of paths and arguments specified in the references. Be very careful of the case and spelling of paths and arguments in your scripts. Tccsds puts the script into an internal database, and then the tccsds components independently fetch the arguments they need. This means that tccsds is unaware if an optional argument is used or not, so it does not warn you if you misspell one. For example, if you misspell the optional packet argument "2hdr" as "Hdr2," tccsds will not tell you that "Hdr2" s incorrect, and it will use the default value for the secondary header option. If you construct a script that does not do what you want, always check the path and argument spellings first.

Here are the paths from the example:

```
   main
   output.plain
   device.file.user0
   stream.pkt.Packets
   stream.pkt.Packets.region.data
```

You might think of a path as a directory tree. Each "dot" subpath refines the subpath to its left, and you might think of a subpath as a subdirectory. Do not insert white space inside a path string, but you may precede it with white pace, and you must separate it from an argument list with white space.

The left-most subpath, the primary path, describes a general class of component, and only a few pre-defined names exist. (Custom applications may dd additional primary paths, but this is unlikely.) Tccsds recognizes the following primary paths:

main	Tccsds demands that one main statement must exist. It defines the scenario type, which is cadu, v1tf, or tc.
output	Tccsds demands that one output statement must exist. It defines the format and device of a scenario's output file.
device	Each device defines one input or output device. This could be a disk file, tape, or port.
stream	Each stream defines a component that handles or makes units.
event	Each event defines when something will happen based upon unit ids.

The next subpath to the right of the primary one, the secondary path, refines the primary path. Tccsds recognizes only certain secondary paths, and your choices depend on the primary path. For example, event accepts only unit or value as a secondary path. For streams, the secondary path is the stream type. or example, "stream.pkt" defines a version one packet stream.

Subsequent subpaths further refine the path. For some paths, subsequent subpaths are pre-defined, and in other cases the user may define any label the wishes. It always depends on the subpaths that precede it.

The argument list, if it exists, follows the path and is separated from it by white space. Every path at every level has a distinct, acceptable argument list, if any. Consult the reference material for the valid arguments. Separate arguments with white space.

Each argument consists of a label immediately followed by a value field, which must be enclosed in parentheses. The value field may not be blank. Do not separate the argument label from the value with white space. For example,

"inStream(ap101)" is legal, and "inStream (ap101)" is not.

The case and spelling of the argument label must exactly match the case and spelling as defined in the references. If a value field has only certain allowed string values, then it too must match in case and spelling.

Depending on the argument label, a value field may be a string or a decimal or hexadecimal number. If an argument expects a hexadecimal number, you may enter t as-is, or you may include a "0x" prefix. For example, the "fill" argument requires a hexadecimal number. If you wanted to change the fill to hexadecimal F7, you could write either fill(f7) or fill(0xf7). In the sample scripts and examples, we always use the "0x" prefix to clearly show the value s hexadecimal and not decimal.

The user either picks string value fields from a pre-defined list, or he may enter any string of his choice. It depends on the argument which one is expected. For example, the user type pattern loader lets the user define in the script a string that tccsds will insert into the data region of packets. his string may include almost any character including spaces. (This is one of the rare occasions when white space is not treated as a separator.) However, do not put parenthesis inside a value string because it will confuse the parser.

Let's examine the sample script line by line. The first statement is:

```
main c(cadu)
```

CCSDS demands that main exists, and it may be anywhere in the script. It has no mandatory argument. The value tells us that this is a CADU scenario as opposed to a V1TF or TC one. The scenario type tells tccsds which stream types re allowed in the script. Since we are creating nothing but packets in this script and packets are in all three scenario types, we could have used cadu, 1tf, or tc in this particular scenario.

Here are the allowed stream types for the three scenarios. The subpath is always the secondary path in a stream definition as in "stream.pkt." For custom applications, there are often additional types.

CADU -	Telemetry
pkt	Version one packets
pktCrate	Version one packets
cadu	CADU frames

V1TF -	Version one transfer frame telemetry
pkt	Version one packets
pktCrate	Version one packets
segpkt	Segmented packets
v1tf	Version one transfer frames
master	Version one transfer frames master channel

TC -	Telecommanding
pkt	Version one packets
pktCrate	Version one packets
tcseg	Telecommand segments
tctf	Telecommand transfer frames
cltu	Command link transmission units
tcphy	Telecommand physical channel

The following stream types are Tiger core streams and are available in every scenario:

subpath	description
mux	Interleaves units from multiple streams into one stream
record	Construct custom units through the script
depot	Reads units from a file
frame	Generic telemetry frame

The next two statements from user0.script describe the output. Tccsds demands that there be one output statement, but other applications may allow more.

```
output.plain device(user0) inStream(Packets) max(0)
device.file.user0 name(user00.dat) access(w)
```

The secondary path describes the output format. You may choose from plain, fancy, files, sim, stgen, and tdg. The sim, stgen, and tdg formats are for specific GSFC Code 521 boards, which transmit data to test systems. The plain, fancy, and files formats have no specific target transmitter.
In the plain format, which we are using in the example, tccsds writes units back-to-back to a file with no unit header or trailer or file header or trailer. The file is in binary format, and you will need to use a dump tool such as the UNIX od to view it.

In the fancy format, tccsds writes the units in binary to a file, but each nit has a header, and sometimes tccsds writes two copies of a unit, one with errors and one without. There is also a file header. Use the fancy format when you are running a scenario in stages (i.e., separate runs of tccsds) and wish o create intermediate files. The fancy format retains complete unit information, which the plain format does not.

The files format is a special enhancement of plain or fancy. With the files type, you specify a maximum file size, and tccsds creates a set of files. No file in the set is bigger than the maximum size.

In the user0 example, we will create a plain format file. The device(user0) argument describes the kind of output device. The value is a label to a device statement. Every output must be attached to at least one device. The output module will get units to write to the device from the stream described by nStream(Packets). "Packets" is a stream label. Every output must link to a stream. The last argument, which is optional, defines the maximum number of nits that the output module will write. The default is zero, which means unlimited, so we could have omitted this argument. In this case, the output module will continue writing units to the device until the input streams provides no more. We have specified a maximum in the packet stream description, but we could have just as easily specified the maximum in the output description. Note that the other output types have different arguments, but every type must attach to an input stream and to at least one output device.

Device is a Tiger primary path. It defines a kind of device, and Tiger uses it for either input or output. The secondary path defines the device type. Currently, Tiger supports two types in its core library: file and null. However, there may be additions, so check the references. Other device types might include tapes and sockets.

The third subpath is the device label. You may choose any name for a device provided the name is different from any other device label. You use the label o link the device to other components. In the user0 scenario, we name the file device "user0," and we specified device(user0) in the output statement to link it to the device. A device may be linked to only one component.

We are defining a file device. The file device uses the UNIX or MSDOS file naming conventions, and it is almost always a disk file. The name(user00.dat) argument says that our output file will be named user00.dat. The access(w) argument says we are writing to the device. If it were access(r), then tccsds would expect to read the device. All devices require the access argument, but the other arguments depend on the device type.

The null device discards all records when used as an output device. This device is useful when we are first writing a script and we want to quickly test it without creating an actual output file. Typically, a user would use the null device in his script to only get the expected results file. Once satisfied with the results, change the device to a file device, and make the final run.

When used as an output device, you must specify access(w). When used as an input device, you must specify access(r), and you must specify the length of the file in bytes for which the null device is substituting. For example, if you are using the null device for an input file that is 62,410 bytes long, then you must specify length(62410) in the null device statement. If you use the wrong null, then your expected results file will be inaccurate. The null device cannot be used as a substitute for all input devices. You only may use it to substitute for a raw file pattern loader. (See references.)

The next statements in user0.script define a packet stream.

```
stream.pkt.Packets max(1000) appid(10) tc(0) length(50) 2hdr(0) checksum(0)
stream.pkt.Packets.region.data type(F) pattern(0x10)
```

A stream is a module that makes or handles units. A stream are the most common component in a script.

Some streams act like factories because they create units of a particular type. For example, in CCSDS we have packet streams and CADU streams, which respectively make packets and CADUs. Another example of a factory stream is the record stream, which is a generic core stream. The record stream lets you design and create custom units through the script.

Other streams read units from a file. For example, the depot stream, which is generic core stream, reads units from a plain or fancy file. In addition, almost every CCSDS factory stream may be commanded through the script to act as a depot instead.

Finally, there are utility streams such as the mux stream. The mux stream merges multiple input streams into one output stream. In a CCSDS CADU scenario, we would use a mux to interleave the CADUs from a group of virtual channel CADU streams. We discuss the mux stream in more detail in another section.

The user0 example creates a CCSDS version one packet stream. The setup says it will create 1,000 packets for application id 10. They will be telemetry packets with no secondary header. All packets will bethe same length, which is 50 bytes long including the primary packet header. In every packet data region, it will repeat the pattern 0x10. Here are the statements once again:

```
stream.pkt.Packets max(1000) appid(10) tc(0) length(50) 2hdr(0) checksum(0)
stream.pkt.Packets.region.data type(F) pattern(0x10)
```

The secondary path, which is "pkt," identifies this as a CCSDS packet stream. the third subpath is a label. We chose "Packets," but we could have picked any name as long as it was different from any other stream name. The label is for identification and linking. Notice that the output module is getting its packets from inStream(Packets). The label links it to this packet stream.

The argument max(1000) restricts the stream to making at most 1,000 packets. The default to this argument is zero, which means make unlimited packets. If we used max(0), then this stream would have worked indefinitely or until some other condition caused it to stop.

The argument appid(10) says this packet stream is to make packets for application id 10, which it inserts in that field of every packet. In a typical scenario, we would define a unique packet stream for every application id. However, this is a practice and not a restriction. If needed, you may create more than one packet stream with the same application id. The only difference between them need be the label.

The tc(0) argument says that this is not a telecommanding packet. That is the default, and we could have omitted this argument. If this were a telecommanding scenario (main c(tc)), then we would want to set this field to tc(1).

The length(50) argument says that all packets are to be 50 bytes long, which includes the packet primary header. We could have chosen to create variable length packets. This requires different arguments and additional statements; we will show an example in another section.

The 2hdr(0) argument says we do not want a secondary header in our packets. If we had set 2hdr(1), then the stream would have put a secondary header in every packet. We would then need to define the secondary header length, and we would need a statement to define the secondary header region. Also, we could have specified an event as in 2hdr(event1). We would do this if we wanted to insert secondary header in selected packets. We will show an example later after we have discussed events.

The checksum(0) argument says we do not want a checksum in every packet, which is the default. We could have omitted this argument. If we had set checksum(1), then the packet stream would compute a checksum for each packet by adding all bytes in the packet minus the last one (modulo 256), and it would put it in the last byte of the packet. This can be a useful verification tool. After passing through a target processing system, one could recompute and compare the checksums to check packet integrity.

The second packet stream statement introduces an important stream concept: the region. A region is an area within a unit that is defined by a name, a start it, and a bit length. A unit may have any number of regions. They may intersect, and they may change in location and size from unit to unit. Each unit type recognizes a predefined set of regions. For example, a packet may have a data region and a secondary header region. The existence of any region depends on stream arguments. In the user0 example, we have a data region but no secondary header region. The importance of regions is that Tiger provides several standard tools to fill them.

If a stream fills a region, then you must provide a region definition statement for it. The statement tells the stream how it will fill the region. To compose a region definition, add the subpaths "region" and the region name o the stream path. For example, the data region path for our packet stream is
"stream.pkt.Packets.region.data."

If we had a secondary header, that path would be "stream.pkt.Packets.region.2hdr." The region names are predefined by the particular stream. (See the reference documentation.)

The data region definition for our packet stream is this:

```
stream.pkt.Packets.region.data type(F) pattern(0x10)
```

The type(F) argument says to fill the region with a repeating byte pattern. The pattern(0x10) says use 0x10 as the fill byte. Since we do not have a secondary header or checksum, the data region is the entire packet minus the first six bytes, which is the primary header. If we had a secondary header, the data region would begin with the first byte after the secondary header. Since our packet definition is complete, we can show what the first packet would look like. Here is an ASCII dump in hexadecimal of the first packet:

```
00000000:  000AC000 002B1010 10101010 10101010
00000010:  10101010 10101010 10101010 10101010
00000020:  10101010 10101010 10101010 10101010
00000030:  1010
```

Suppose we wanted a secondary header. As an example, let us rewrite the packet stream so that we insert a seven byte secondary in every packet. Normally we would put an incrementing timecode in the secondary header region, but for his example we will use the byte pattern 0xdd. Here is the complete packet stream definition. Notice that we now have two region definitions. The statement order does not matter. We will also enable the packet checksum in his example.

```
stream.pkt.Packets max(1000) appid(10) tc(0) length(50) 2hdr(1) -
          2hdrLength(7) checksum(1)
stream.pkt.Packets.region.data type(F) pattern(0x10)
stream.pkt.Packets.region.2hdr type(F) pattern(0xdd)
```

The first packet from this setup will look like this:

```
00000000:  080AC000 002BDDDD DDDDDDDD DD101010
00000010:  10101010 10101010 10101010 10101010
00000020:  10101010 10101010 10101010 10101010
00000030:  105A
```

The Tiger core provides eight different ways to fill a region. Tccsds adds two more to insert CCSDS timecodes. Other applications may add even more. The region fillers are called patterns or pattern loaders.

A region statement's arguments depend on the pattern. The type(name) argument, which is common to all region statements, defines the region type. Here is a brief explanation of every pattern loader in the Tiger core. See the core references for full details. The type is the argument value for the type(name) argument.

Fixed Pattern (type F)
As shown in the simple example, this loader repeats the byte pattern in all bytes in the region.

```
stream.pkt.Packets.region.data type(F) pattern(0x10)
```

Random Pattern (type A)
The loader fills the region with random bytes. However, if you rerun the scenario on the same computer, you will get the same results.

```
stream.pkt.Packets.region.data type(A)
```

Step Pattern (type S)
The loader fills the region with a step or ramp pattern. It continues from where it left off when it fills the next region. You specify a start byte value in decimal, a repeat count, and a step value. The example would generate the following string in a region: 01010101 03030303 05050505 07070707 09090909

```
stream.pkt.Packets.region.data type(S) start(1) step(2) repeat(3)
```

User Pattern (type U)
The loader fills the region with a string of values that are specified in the script. For this pattern, you must add statements that define the values. You may specify the values in decimal, hexadecimal, or text, but you may use only one format.

Raw File Pattern (type R)
The loader fills the region with data that it reads from a file. It continues reading bytes from the file to fill subsequent regions. When it exhausts the file, it usually shuts down the stream, but it may continue filling more regions with constant fill. The raw file pattern loader is a good way to fill packets or other units with meaningful data.

Empty Pattern (type E)
The loader does nothing; the region is not filled. Most streams do not allow this type. It is useful when more than one stream is constructing a unit, and one stream must reserve space that a subsequent stream will fill. This happens in the V1TF scenario. The Master Channel stream fills empty regions that the 1TF streams prepare.

Consumer Pattern (type C)
The loader gets units from an input stream and packs them into a target unit's region. It can either put integral units per region or it can stack them back-to-back, splitting units across regions if necessary. In the V1TF and CADU scenarios, the frame stream uses this loader to put packets into its frames.

Inlay Pattern (type I)
This loader is similar to the consumer pattern except it puts only one input unit per region, so the region must be at least as big as the input unit.

Under CCSDS, the following timecode patterns are available. They are almost always used in the packet secondary header. The region lengths depend on timecode setup options. Each timecode automatically increments by the step value every time the loader fills a region. See the CCSDS blue book on time codes for format information. The pattern descriptions are in cscript.doc.

Day Segmented Timecode (type tcday)
Unsegmented Timecode (type tccuc)

5.2 TIGER TUTORIAL: MAKING CADUS

In the next example, we will expand the simple script by inserting a CADU stream into the data flow. Packets will flow into a CADU stream, which fill flow into the output module. We will also show some additional options, such as idle units and events.

```
-------------
# user1.script
# This CADU script makes 100 CADUs from a single packet stream.

main c(cadu)

device.file.simple1 name(simple1.dat) access(w)
output.plain device(simple1) inStream(frames) max(100)
#output.stgen inStream(frames) device(simple1) recordSize(1000) max(100)

stream.cadu.frames service(P) length(1000) vcid(0) spid(100) crc(1) -
          frameSync(0x1acffc1d) idleEvent(idleFrame)
stream.cadu.frames.region.data type(C) inStream(ap10) lastUnit(I) -
              ERcomposition(1)
event.unit.idleFrame.range uid0(1) uid1(4)
```

```
stream.pkt.ap10 appid(10) length(500) 2hdr(1) 2hdrLength(7) idleEvent(iap10)
stream.pkt.ap10.region.data type(U) wrap(0) format(S)
stream.pkt.ap10.region.data.data s0(This is text packet data for appid 10. )
stream.pkt.ap10.region.data.data s0(vcid=0 spid=100 CRC=on. )
stream.pkt.ap10.region.2hdr type(tcday) day(3000) 16BitsDay(1) msStep(100)
event.value.iap10.recur start(10) skip(99) v(50)
-------------
```

This is a CADU scenario. The output statement says we are writing 100 units to the "simple1" device, which is the plain file simple1.dat. This time the input stream is "frames," which is a CADU stream.

We have included but commented out an alternate output. In the alternate, we are creating an STGEN format output file. The STGEN format is for a GSFC Code 521 simulator board. The recordSize(1000) argument tells the output module that the length of the records from "frames" is 1000 bytes, which is our CADU length. The STGEN board has four megabytes of memory for test data, but we are only using 100 records (100,000 bytes) of it in this example.

The following statements define a CADU creation stream for one virtual channel. We would make more of these if we had additional virtual channels. The first statement says the stream is doing the path service. It could also do VCA/VCDU and bitstream services. Each CADU is 1,000 bytes long and is CRC encoded. The stream is making CADUs for virtual channel zero. The spacecraft id is 100. We have specified a sync pattern, but this was unnecessary because we have specified the default. Lastly, the first statement identifies an idle event by name. The idle event tells the CADU stream when it should make idle or fill CADUs. Some target processing systems lose one or more frames at the start of session until they lock on the data, so we would like to start with a few idle frames to ensure the target does not discard any packets.

```
stream.cadu.frames service(P) length(1000) vcid(0) spid(100) crc(1) - frameSync(0x1acffc1d) idleEvent(idleFrame)
stream.cadu.frames.region.data type(C) inStream(ap10) lastUnit(I) - ERcomposition(1)
event.unit.idleFrame.range uid0(1) uid1(4)
```

CADU has three regions: a data region, an OCF (CLCW) region, and an insert one region. We have not enabled the OCF or insert zone, so we must supply a region statement for the CADU data region only. The second statement is the data region definition. It is type C, a consumer region, which means the CADU stream will be packing units back-to-back into the data region. The input stream is the "ap10" packet stream, so the CADU stream will be loading packets, splitting them across CADUs if necessary.

The lastUnit(I) argument says to put idle packets in the last CADU to fill it put in case we run out of packets early. By default, the consumer loader fills with a constant byte pattern. The ap10 packet stream does not shut down early in our example, but it is a good idea to provide the lastUnit(I) argument in case we later change how we make packets.

The ERcomposition(1) argument tells the stream to write CADU composition information to the expected results file. This output can get quite lengthy, so it is off by default. However, it is also quite useful because it shows which packets are inside every CADU.

The third statement is an event statement, which the CADU stream references. Events are unique entities that define when something should happen. Any number of streams may reference the same event. The CADU stream uses this event to tell it when it should generate idle CADUs. The event is a unit event, which means it gives only true and false information. It is a range event, and the range is unit one through four inclusive. To the CADU stream, it means it should make idle CADUs for the first four frames, frames #1 through #4.

The CADU stream get packets from "ap10." This is our packet stream. It is making packets for application id 10 with a constant total length of 500 bytes. It is putting a seven byte secondary header in each packet, and it is also making a few idle packets.

```
stream.pkt.ap10 appid(10) length(500) 2hdr(1) 2hdrLength(7) idleEvent(iap10)
```

Since we turned on the secondary header, we must specify a secondary header region. Secondary headers are often timecodes, so we choose the CCSDS Day segmented timecode. We need at least seven bytes because the Day Segmented options demand it. If we had made the region larger, the stream would put zeroes in the extra trailing bytes. We start the time at day 3,000 from epoch, and we set the step size to 100 milliseconds. Every time the stream loads a secondary header, the timecode pattern automatically increments by 100 milliseconds.

```
stream.pkt.ap10.region.2hdr type(tcday) day(3000) 16BitsDay(1) msStep(100)
```

To fill the packet data region, we have selected the user pattern. This pattern demands that we provide the data pattern in the script, so we must specify one or more "data" subpaths to the user region path. We want to enter the data pattern as text, so we set format(S), which means "text string." Our other choices are format(H) for hexadecimal and format(D) for decimal.

```
stream.pkt.ap10.region.data type(U) wrap(0) format(S)
stream.pkt.ap10.region.data.data s0(This is text packet data for appid 10. )
stream.pkt.ap10.region.data.data s0(vcid=0 spid=100 CRC=on. )
```

The "data" subpaths may have arguments s0() through s9(), and we may have more than one statement. For the D and H formats, each "s" argument specifies one byte. The user pattern constructs an array of values by concatenating all "s" arguments into one list. The order depends on first the order of the data statements and second on the order of the "s" arguments, s0() through s9(). If n "s" argument is missing, it simply skips it. In the example, the user pattern constructs this string:

```
"This is text packet data for appid 10. vcid=0 spid=100 CRC=on. "
```

It will repeatedly copy this string to the data region until it completely fills t. Normally it will wrap any leftover string into the next region, but the rap(0) argument tells it to discard any leftover and to start every new region with the beginning of the string.

We decided to insert idle packets into the packet stream, so we included the idleEvent(iap10) argument, and we provided an event statement. This statement tells the packet stream to make a fifty byte idle packet and to insert it at unit ten and repeat it every 100th packet thereafter.

```
event.value.iap10.recur start(10) skip(99) v(50)
```

This is an example of a value event because it has a value field with each event statement in addition to the true and false information that a unit event provides. Streams that only need the true/false information can still use it; they simply ignore the value. For example, we could have used it in the CADU stream to define when it should construct idle CADUs. It would have made 1,000 byte idle CADUs every 100th CADU, and it would have ignored the v(50) argument.

A unit event is actually a degenerate value event. If used as a value event, a unit event gives a value of one. If we had set idleEvent(idleFrame) in the packet stream definition, the packet stream would have tried to create one byte packets, which is illegal and would have caused an error.

The example event is also a recurrent event. Both unit and value events may be either recurrent or range events. A recurrent event defines a pattern that is repeated again and again.

5.3 TIGER TUTORIAL: ERRORS

So far, we have created only clean data files. In this example, we will insert errors into some of the packets and CADUs. This is a CADU scenario. We write plain units (CADUs) to the file "user2.dat."

```
-------------
# user2.script
# This CADU script makes 100 CADUs from a single packet stream.
# In this example, we insert errors into both streams.

main c(cadu)

device.file.user2 name(user2.dat) access(w)
output.plain device(user2) inStream(vc0) max(100)

stream.cadu.vc0 service(P) vcid(0) spid(100) idleEvent(vc0i) drop(drop) -
            RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc0.region.data type(C) inStream(ap10) lastUnit(I) ERcomposition(1)
stream.cadu.vc0.error.set convey(1) label(VCerror) event(vc0e1) -
           startbit(42) bits(6) v(60)
stream.cadu.vc0.error.set convey(0) label(SCerror) event(vc0e2) -
           startbit(34) bits(8)

event.unit.drop.range uid0(25) uid1(25)
event.unit.vc0e1.range uid0(90) uid1(90)
event.value.vc0e2.range uid0(90) uid1(90) v(77)
event.value.vc0e2.range uid0(92) uid1(93) v(55)
event.unit.vc0i.range uid0(1) uid1(1)
event.unit.vc0i.range uid0(100) uid1(100)

stream.pkt.ap10 appid(10) length(500) 2hdr(1) 2hdrLength(7)
stream.pkt.ap10.region.data type(U) wrap(0) format(S)
stream.pkt.ap10.region.data.data s0(This is text packet data for appid 10. )
stream.pkt.ap10.region.data.data s0(vcid=0 spid=100 RS=4,255. )
stream.pkt.ap10.region.2hdr type(tcday) day(3000) 16BitsDay(1) msStep(100)
stream.pkt.ap10.error.add convey(0) label(+1seq) event(ap10e1) -
           startbit(18) bits(14) v(1)
stream.pkt.ap10.error.flip convey(0) label(appidErr) event(ap10e2) -
           startbit(5) bits(1)
event.unit.ap10e1.range uid0(2) uid1(5)
event.unit.ap10e2.range uid0(10) uid1(13)
event.unit.ap10e2.recur start(50) repeat(1) skip(4) occur(3)
-------------
```

In the "vc0" CADU stream, we are Reed-Solomon (RS) encoding the frames instead of CRC encoding them. RSencode(1) enables RS encoding and automatically disables CRC encoding by default. (It is legal, but unusual, to have both encoding methods enabled simultaneously. The RS parity would follow the CRC parity in the frame.) RSinterleave(4) sets the interleave to four in this example. Popular choices are one through five. RScodeLength(255) is the codeword length. The maximum value is 255.

There are two additional RS arguments for which we used the defaults. RSdual(1) enables or disables dual mode. The CCSDS standard uses dual mode, so this argument should always be enabled (and is by default). RSempty(0) is another boolean argument. When enabled, it leaves space for the RS parity, but it does not compute it. RSempty(0) is the default, which is to compute the

parity. RS encoding is expensive in terms of CPU time. For interleave four and codeword length 255, over 80% of tccsds' time is in RS encoding. You might choose to set RSempty(1) when you are dry running a script or you are transmitting the test file through hardware that does the encoding for you.

Notice that we did not specify the length(n) argument. When RS encoding, the frame length derives from the RS parameters. If we specified a length, tccsds would either ignore it or would warn us if it did not match the computed value. To compute the frame length when RS encoding, multiply the interleave by the codeword length and add four for the synchronization pattern. In our example the frame length is 1024 (4 X 255 + 4).

Most spacecrafts desire a frame length that is a multiple of four, and the requirements may specify "virtual fill" to achieve it. To handle virtual fill, simply subtract the number of bytes of virtual fill from the maximum codeword length (255) and use that as the codeword length value. For example, a popular frame length is 256 bytes at interleave one and three bytes of virtual fill. To create these CADUs, set RSinterleave(1) and RScodeLength(252) (1 X 252 + 4).

In the "vc0" CADU stream, we are demonstrating a more complicated use of an event. We have specified idleEvent(vc0i), which identifies when we want to insert idle CADUs into the stream. The "vc0i" event has two statements:

```
event.unit.vc0i.range uid0(1) uid1(1)
event.unit.vc0i.range uid0(100) uid1(100)
```

In this case, we want CADUs #1 and #100 to be idle. Notice that we may create composite events by writing more than one event statement with the same name. An event may consist of any number of range and recurrent statements. For value events, each statement may have its own v(n) value argument.

Sometimes it is important to know the order in which tccsds checks the statements in an event. This is especially true when each event statement has its own value argument. Tccsds always checks range events before recur events no matter how they listed in the script. Within range or recur events, tccsds checks them in the order that they are listed in the script. To avoid confusing a reader, alway put range statements ahead of recur statements in an event definition.

In the "vc0" CADU stream, we decided to discard the 25th frame, which will create a virtual channel gap and a packet sequence gap. To accomplish this, we have specified the drop(vc0d) argument. "vc0d" is a ranged, unit event, and it specifies the range 25 to 25. The drop argument is available to all streams and is not just a CADU stream argument.

The effects of dropping CADU #25 are reflected in the expected results file for this scenario. (See caduSample.er for an annotated expected results file.) Here are the significant lines:

```
c      1  uid=24 fhp=52 2.39 2.40 2.41 last=332
c      1  uid=25 fhp=168 2.41 2.42 2.43 last=216
drop 1  uid=25
c      1  uid=26 fhp=284 2.43 2.44 2.45 lsat=100
gap  1   uid=26 actual=24 expected=23 gap=1
```

Packet #41 will be a partial packet. The first 332 bytes will be available, but the last 168 bytes will be missing. Packet #42 will be missing. Packet #43 is also partial, but here the first 216 bytes will be missing, which includes the packet header. The target system will probably be able to handle packet #41, but packet #43 is likely a lost cause because the primary header is missing. The net result is probably a gap of two packets. Note that the expected results file predicts a CADU gap but not a packet gap. The reason is we made the gap in the CADU stream, so the packet stream is unaware of data loss. (It is too late for it to report it.) Also, the CADU stream uses core tools to fill its regions, so it knows very little about the input units. The "actual" and "expected" numbers in the CADU gap report are virtual channel sequence numbers.

To introduce bit errors into a stream, we use error statements. Append "error" and an error type to the stream path. There are four different error types:

- set deposits a 1-32 bit value in a unit. The new value replaces the old one.
- add adds a 1-32 bit value to an existing 1-32 bit value in a unit.
- flip inverts any number of contiguous bits in a unit.
- flipmask inverts 1-32 bits in a unit controlled by a 32 bit mask. Ones in the mask designate bits to be flipped. Zeroes in the mask designate bits not to be flipped.

We are inserting two different errors in the CADU stream. Each error consist of an error statement and an event. (We do not need a unique event for every error. One event may may be referenced by multiple components.) Here is the first error specification:

```
stream.cadu.vc0.error.set convey(1) label(VCerror) event(vc0e1) - startbit(42) bits(6) v(60)
event.unit.vc0e1.range uid0(90) uid1(90)
```

These statements tell tccsds to deposit the value 60 in the 90th CADU starting at bit 42 and extending for six bits. (Bit zero is the first bit and corresponds to the first bit of the synchronization pattern.) Bits 42-47 is the virtual channel id, so we are overwriting the CADU's valid virtual channel with virtual channel 60.

The label(VCerror) argument has no control function and is merely an explantory message that the stream writes to the expected results file.

The convey(n) argument is a boolean argument that tells tccsds whether or not to convey the error onto the next receiving stream. When on (convey(1)), the receiving stream and all subsequent receiving streams would show the error in the appropriate unit. Each stream applies the error AFTER doing any encoding. For example, suppose a packet stream fed packets into a CADU stream, which fed CADUs into a NASCOM block stream. Also suppose the CADU stream only did CRC encoding. Now suppose we insert a bit error in the tenth packet. Then the expected results file would mark the affected packet and any CADU or NASCOM block containing any portion of the packet as having an error. Furthermore, the target system should report a CRC in the frame and a poly error in the NASCOM block that contained the bit error because errors are applied after encoding when convey(1) is set.
When off (convey(0)), the stream hides the error, and all subsequent streams behave as if no error had occurred. At the packet level, the behavior is as if the error occurred at the instrument. The streams apply errors BEFORE they do any encoding. This means if the encoding algorithm is a correcting one, such as Reed-Solomon encoding, then the target system's RS decoder will not correct the error because it won't believe there is one.

So in the example, will the target system see virtual channel 60 in the 90th CADU? The answer is no, assuming the target has a RS decoder. Since we have set convey(1), the stream applied the error after it had RS encoded the frame. This means the target system's RS decoder should detect and correct the error, which means subsequent processors will see the correct virtual channel id and not our deposited value.

In general, if the target system has a correcting decoder, then you should almost always set convey(0) in all streams so that the encoder does the encoding after any errors are applied. Otherwise, the decoder will probably fix the error, and your carefully planned error scenario will be wasted. Set convey(1) if your test is check out the decoding functions of the target system.

The second error in "vc0" tells it to deposit 77 into the spacecraft id field, which is bits 34-41, for CADU #90. It also deposits 55 into the spacecraft id field for CADUS #92 and #93. This error is not conveyed, so the Reed-Solomon decoder will not correct it.

```
stream.cadu.vc0.error.set convey(0) label(SCerror) event(vc0e2) - startbit(34) bits(8)
```

```
event.value.vc0e2.range uid0(90) uid1(90) v(77)
event.value.vc0e2.range uid0(92) uid1(93) v(55)
```

Notice that we have omitted the error statement's v(n) argument and instead have used a value event. If an error that expects a value has no v(n) argument, it uses the value provided by the event. Using a value event, we can deposit more than one value using one error statement as we have done in the example. Had we mistakenly used a unit event, then tccsds would have deposited "one" in the spacecraft id, which is a unit event's value.

The packet stream is the same as in the previous example except we now add errors. There are two errors in this stream. Here is the first:

```
stream.pkt.ap10.error.add convey(0) label(+2seq) event(ap10e1) - startbit(18) bits(14) v(1)
event.unit.ap10e1.range uid0(2) uid1(5)
```

For packets two through five inclusive, the stream will add one to whatever is in the packet's sequence field. This will cause two gaps in the sequence count. The error is not conveyed, so the Reed-Solomon decoder will not correct it. If the new sum is greater than the target field's maximum value, the stream removes bits until it fits.

In the second error, the packet stream is using the flip error type. It will invert the most signficant bit of the application id for ten packets, which the event "ap10e2" describes.

```
stream.pkt.ap10.error.flip convey(0) label(appidErr) event(ap10e2) - startbit(5) bits(1)
event.unit.ap10e2.range uid0(10) uid1(13)
event.unit.ap10e2.recur start(50) repeat(1) skip(4) occur(3)
```

This composite event contains both a range and recurrent event. The first statement causes the error to be applied to packets 10-13. The second statement causes the error to be applied to packets 50, 51, 56, 57, and 62, 63.

5.4 TIGER TUTORIAL: MUX

In the previous scenarios, we created CADUs for one virtual channel, and that stream got packets from only one packet stream. In this scenario, we show how to merge CADU streams and packet streams using the mux stream. The mux stream is a Tiger core stream that interleaves units from multiple input streams. In this example script, we merge two CADU streams representing virtual channel 0 and virtual channel 1. For the "vc0" CADU stream, we use a mux stream to get packets from the "ap10" and "ap20" packet streams.

The dataflow in this scenario is as follows: Packet streams "ap10" and "ap20" flow into a mux, which flows into the CADU stream "vc0." Then the CADU streams "vc0" and "vc1" flow into another mux stream, which flows into the output module.

We demonstrate several other new features in this scenario: a VCA/VCDU service CADU stream, the OCF (CLCW) CADU region, variable length packets, and filling a region from a data file.

```
-------------
# user3.script
# This CADU script makes 100 CADUs from multiple CADU and packet streams.
# vcid 0 is path service. Packet streams ap10 and ap20.
# vcid 1 is vca/vcdu service.
# Other scenario features: OCF (CLCW) in CADU and variable length packets.

main c(cadu)

device.file.user3 name(user3.dat) access(w)
output.plain device(user3) inStream(vc) max(100)
```

```
stream.mux.vc default(vc0) idle(vc0) eos(S) er(1)
stream.mux.vc.range uid0(1) uid1(1) idle(8192)
stream.mux.vc.recur start(1) skip(1) stream(vc0)
stream.mux.vc.recur start(2) skip(1) stream(vc1)

stream.cadu.vc1 service(V) vcid(1) spid(100) RSencode(1) RSinterleave(4) - RScodeLength(255)
ocf(1)
stream.cadu.vc1.region.data type(R) device(vc1file) fill(0) extendPastEOF(0)
stream.cadu.vc1.region.ocf type(U) wrap(0) format(H)
stream.cadu.vc1.region.ocf.data s1(0x01) s2(0xfc) s3(0x30) s4(0x19)
device.file.vc1file name(user3.doc) access(r)

stream.cadu.vc0 service(P) vcid(0) spid(100) RSencode(1) RSinterleave(4) - RScodeLength(255)
stream.cadu.vc0.region.data type(C) inStream(vc0mux) lastUnit(I) - ERcomposition(1)

stream.mux.vc0mux default(ap10) idle(ap10)
stream.mux.vc0mux.recur start(1) repeat(49) span(100) stream(ap10)
stream.mux.vc0mux.recur start(50) repeat(49) span(100) stream(ap20)

stream.pkt.ap10 appid(10) length(500) 2hdr(1) 2hdrLength(7)
stream.pkt.ap10.region.data type(U) wrap(0) format(S)
stream.pkt.ap10.region.data.data s0(appid=10 vcid=0 spid=100. )
stream.pkt.ap10.region.2hdr type(tcday) day(3000) 16BitsDay(1) msStep(100)

stream.pkt.ap20 appid(20) variableLength(1) varLenEvent(var20) 2hdr(1) - 2hdrLength(7)
stream.pkt.ap20.region.data type(U) wrap(0) format(S)
stream.pkt.ap20.region.data.data s0(appid=20 vcid=0 spid=100. )
stream.pkt.ap20.region.2hdr type(tcday) day(3000) 16BitsDay(1) msStep(200)
event.value.var20.recur v(120) start(1) span(5)
event.value.var20.recur v(291) start(2) span(5) repeat(1)
event.value.var20.recur v(816) start(4) span(5)
event.value.var20.recur v(620) start(5) span(5) occur(8)
event.value.var20 default(64)
-------------
```

The output module in this scenario gets units from the "vc" stream, which is a mux stream and not a CADU stream. No stream other than a mux can get units from multiple input streams, so you must use a mux to merge them first. We use two mux streams in this scenario. In the first one, we are interleaving CADUs from two different CADU streams, each one representing a different virtual channel.

Note: Although we define a separate CADU stream for each virtual channel and a separate packet stream for each application id, this is a practical application and not a required one. You may even construct streams that are identical in all respects. Remember, though, that each stream is an independent entity that is unaware of any duplicate's behavior.

The primary script setups for a mux are the names of the input streams and how it should interleave their units. The mux uses the event architecture to do this. The "vc" mux definition says to make the first CADU an idle one and then to alternate between the two virtual channels.

The range statement causes the first CADU to be idle. The idle(8192) argument defines the idle CADU. The value is the unit length in bits. Since CADUs are all the same size, tccsds ignores the value. However, a value is required, so we must enter some number. The actual value in this case does not matter.

The third and fourth statements define input streams and when the mux should get their units. In this case, the mux will get CADUs from "vc0" on all odd numbered CADUs and from "vc1" on all even numbered CADUs. Although the "vc0" statement says to start "vc0" with the first unit, the

mux will make an idle for the first unit because the range statement has precedence over any recur statement.

```
stream.mux.vc default(vc0) idle(vc0) eos(S) er(1)
stream.mux.vc.range uid0(1) uid1(1) idle(8192)
stream.mux.vc.recur start(1) skip(1) stream(vc0)
stream.mux.vc.recur start(2) skip(1) stream(vc1)
```

Although we show one statement per CADU stream, it is legal to use the same stream name in more than one statement. We would do this to make addition unit definitions for an input stream.

The "stream.mux.vc" statement shows several mux arguments. The default(vc0) argument tells the mux to use "vc0" if for some reason a unit id is not in one of the range or recur statements. The mux will also use the default stream if an input stream does not give a unit when asked and eos(S) is set. This happens in our example for "vc1" because that stream only creates a limited number of CADUs. When the mux exhausts the supply of "vc1" CADUs, it will switch to the default ("vc0") whenever it is supposed to get a "vc1" CADU. If the default stream runs out of units or the default() argument is undefined when needed, then the mux will terminate, which probably will end the scenario.

The eos(S) argument tells the mux what to do if any input stream does not give a unit. The "S" value tells the mux to skip the stream and to use the default stream for a substitute unit instead. Other choices are "I" and "X." "I" causes the mux to request an idle unit as a substitute. "X" causes the mux to terminate immediately.

The idle(vc0) argument tells the mux where to go to get idle units if needed. In the example script, the mux will ask "vc0" to provide idle CADUs. In most cases, any input stream of the right type will suffice. It does not even have to provide regular units to the mux, and you may create a stream just to provide them. In the example,we could have selected "vc1" just as easily as "vc0." The fact that a stream has been asked to provide idles does not affect its normal unit production. For CADUs, all idles will be virtual channel 63 regardless of the provider's virtual channel.

Warning: Sometimes it DOES matter who provides idles. If a CADU stream has an insert zone, then all idles it creates will also have an insert zone. Also, if a CADU stream has the replay flag on, then all idles it creates will also have the replay flag on.

The er(1) argument causes the mux to write information to the expected results file. The file will show exactly how the units have been interleaved. This option can create a large file, so the option is off by default. You may also supply an event name as a value to er(). The mux will then write information only for the selected units.

The technique of using event statements to define a mux's interleave strategy is called strategy A. For complex scenarios, the number of mux event statements can get relatively large. The mux allows a second strategy, strategy F, to define the interleave order. Under strategy F, the mux gets the interleave order from a text file, which simply lists streams in the correct interleave order. Using any text editor, write one stream name per line, left-justfied. To insert an idle unit, insert the word "idle" on a line by itself. When the mux reaches the end of the file, it restarts from the beginning. To use strategy F, specify strategy(F) and a file argument in the "stream.mux.name" statement. See the mux reference for details.

The muxtool program is a tool that creates mux strategy F files. You first create a small muxtool input file that lists stream names and the percentage of their occurrence. (See muxtool.doc for format information.) Then run muxtool with the file, and it creates the strategy F file. For example, suppose you wanted to interleave three virtual channels so that the streams would be distributed as follows:

```
vc0   25%
vc1   60%
```

vc2   14%
idle   1%

You would create a small muxtool input file with these numbers. Muxtool would then make a 100-line file (100 by default but you may create different length spans) such that the idles and streams would be evenly distributed over the span. You could then edit the strategy F file to add or remove entries.

The "vc1" CADU stream performs the VCA/VCDU frame service, so it does not get packets from packet streams. The tccsds program combines the VCA and VCDU services because they are identical at the generation end. The bitstream service (service(B)) is very similar. The primary difference between the VCA/VCDU service and the path service setup is that we attach the data region to a different pattern loader and not to a packet stream.

```
stream.cadu.vc1 service(V) vcid(1) spid(100) RSencode(1) RSinterleave(4) - RScodeLength(255) ocf(1)
stream.cadu.vc1.region.data type(R) device(vc1file) fill(0) extendPastEOF(0)
stream.cadu.vc1.region.ocf type(U) wrap(0) format(H)
stream.cadu.vc1.region.ocf.data s1(0x01) s2(0xfc) s3(0x30) s4(0x19)
device.file.vc1file name(user3.doc) access(r)
```

We have selected a file (called a raw file in the reference) to be the filler for the CADU data region. The file may be text or binary, and for the example we have chosen this document. The CADU stream will create an ordered list of CADUs with each one containing a fragment of this document in the data region. A target system should be able to reconstruct our original file from the CADUs.

Since a file has finite length, streams using a raw file loader usually make a finite number of units. It is likely that the loader will not exactly fill the last CADU. If that happens, it writes a constant fill pattern to finish the last region. The fill(0) argument tells the loader to fill out the last region with zeroes. (The default fill is 0xc9.) The extendPastEOF(0) tells the loader not to continue making CADUs after it has finished the file. When it has consumed the entire document, it will stop. If we had enabled extendPastEOF, then the stream would have continued making CADUs. The data region of every extra CADU would be all zeroes, which we specified in fill(0).

The ocf(1) argument tells the CADU stream that it should put the Operational Control Field, which is a command echo and also known as CLCW, in every CADU. The tccsds program treats the four byte OCF as another region, so we must specify an OCF region statement. Currently there is no special OCF pattern, so we must choose one of the standard patterns. In the example, we have decided to fill each OCF with the user pattern 0x01fc3019. If you need to create a more realistic OCF that changes with time, the best solution is to create a binary file of consecutive, four-byte OCFs. Then use the raw file pattern as we did in the data region to fill the OCF region.

Note that the OCF is dedicated to one virtual channel in this scenario. There is no CADU stream, like the master channel stream in version one transfer frames, that inserts OCFs into the CADUs after they have been interleaved into one stream. However, it can be done by using the record stream. See the record stream references for more information.

The "vc0" CADU stream is almost the same as our other examples except it get packets from a mux instead of a packet stream directly. The setup for this packet mux is similar to the first one we examined.

```
stream.mux.vc0mux default(ap10) idle(ap10)
stream.mux.vc0mux.recur start(1) repeat(49) span(100) stream(ap10)
stream.mux.vc0mux.recur start(50) repeat(49) span(100) stream(ap20)
```

This mux interleaves packets from the "ap10" and "ap20" streams. It toggles between blocks of fifty packets from each one. If it must get an idle packet, it gets it from "ap10." If either packet stream runs out of packets, "vc0mux" goes to "ap10," and if that runs out, it terminates.

The "ap10" packet stream is nearly identical to packet streams we have examined. However, "ap20" has something new. It generates variable length packets.

```
stream.pkt.ap20 appid(20) variableLength(1) varLenEvent(var20) 2hdr(1) - 2hdrLength(7)
stream.pkt.ap20.region.data type(U) wrap(0) format(S)
stream.pkt.ap20.region.data.data s0(appid=20 vcid=0 spid=100. )
stream.pkt.ap20.region.2hdr type(tcday) day(3000) 16BitsDay(1) msStep(200)
event.value.var20.recur v(120) start(1) span(5)
event.value.var20.recur v(291) start(2) span(5) repeat(1)
event.value.var20.recur v(816) start(4) span(5)
event.value.var20.recur v(620) start(5) span(5) occur(8)
event.value.var20 default(64)
```

The variableLength(1) argument identifies variable length packets, and varLenEvent(var20) identifies a value event whose values are the packet lengths. In most of the events that we have used so far, the users have been primarily interested in the true/false nature of the event. This is a case where the stream only cares about the values.

The "ap20" stream creates packets in five different lengths. (The length is a total packet length and includes the primary packet header.) The length pattern is 120, 291, 291, 816, and 620 bytes. Notice that there are two packets of 291 bytes in each set of five. Also, the 620 byte packet only occurs eight times, so the stream substitutes the default, 64 bytes, for the 620 byte packet after the eighth one.

SECTION 6
EXAMPLES OF SCRIPTS AND DATA GENERATION

In this Tiger script, we will create 1,000 CADUs, and we will write them to a plain, binary file called "cadulist.dat." This scenario models the following situation:

vc	appid	pktLength
17	256	332
17	257	340
17	258	128
18	259	784
18	260	392
18	261	128
23	262	964
23	263	944
30	265	332
30	266	340
30	267	128
41	320	780
41	321	580
41	322	560
41	323	572
41	324	571
42	67	642
42	68	642
42	69	276
11	11	64
2 frame service		

The packet length includes the six byte primary header. All packets will have a secondary header. We choose to put CCSDS unsegmented time in the secondary header location. It will consist of the optional pfield, two bytes of coarse time, and three bytes of fine time, which makes a six byte secondary header. We will fill each packet with a constant pattern, which will be different for each application id. Finally, we will put a checksum in the last byte of each packet, which is the sum of all previous bytes modulo 256. We could use this checksum in post-processing to verify proper packet construction.

All frames (CADUs) are 1024 bytes long and are Reed-Solomon encoded with interleave 4 and codeword length of 255. Each frame has a four-byte sync pattern of 0x1acffc1d. (This combination of interleave and codeword lengthforces 1020 byte frames. The sync pattern makes it 1024 bytes.) We are not adding the OCF field (CLCW), insert zone, or any inversion or pseudo-noise encoding.

The script shows a number of streams. Each one either constructs or handles a unit (packet, frame, etc). Although we have listed the streams from CADU makers down to packet makers, the order is not important, and you can arrange them in any order you like. Notice that we have linked the streams together using a "stream" or "inStream" field. For example, the vc17 stream links to a vc17 mux, which interleaves packets from the ap256, ap257, and ap258 streams.

A stream definition consists of a "dot string," such as "stream.mux.top"followed by an argument list. The individual fields in the "dot string" are called paths. The "stream" path identifies a mechanism that makes or handles units such as packets. The second path identifies the kind of stream. These are predefined names such as "mux" or "cadu." Case is important. The third path is a user-selected label that uniquely identifies the stream. No other stream may have the same

name. We use the label to link streams together. You may choose any name except "idle" which has special meaning. Also, avoid special characters such as dot, parenthesis, and dash.

The "main" line is mandatory. The single argument "c()" identifies which of the three CCSDS basic scenarios we wish to run. In this example, we are doing the "cadu" scenario. We could also do "v1tf" (version one transfer frames) or "tc" (telecommanding). Each basic scenario has its own collection of unique streams.

There must be one output defined. In the example, the output is a plain binary file. Output is writing up to 1,000 records to the "out" device, and it is getting records from the stream named "main." There are several different output formats that we could have used, and the device line allows us to choose different types of output devices. In the example, we have commented out a substitute device, the null device. The null device discards all records, and we use it to test our script and to get an expected results file before we run the real scenario.

```
main c(cadu)
output.plain device(out) inStream(top) max(1000)
device.file.out name(cadulist.dat) access(w)
device.null.out name(cadulist.dat) access(w)
```

A mux stream merges unit from multiple input streams into one output stream. The following mux stream interleaves CADUs from the virtual channel CADU streams. This example is taken from an actual test. The user wanted to create an interleave pattern for the first 100 frames and then to repeat it thereafter. The mux setup is more complicated than usual, and the user may have instead chosen to have the mux use an input file. (See muxtool.doc to see how that is done.) For examples of simpler muxes, see the packet muxes below.

The name "top" is not particularly significant, and we could have chosen any name as long as it was unique. (Case is important.) Its only purpose is as a tag so that we can link streams together.

The first fourteen frames out of this mux will be (by vcid): idle, 17, 17, 30, 30, 18, 18, 42, 42, 17, 17, 30, 30, 41. This mux consists of a number of recurrent patterns. For example, line #3 tells us vcid 17 starts with the second output frame and repeats once. This means that output frames 2 and 3 will be from the input stream "vc17." The "span(100)" argument says to do it again every 100 frames, so vc17 will occur again at frames 102 and 103, 202 and 203, and so on. Line #4 says that frames 4 and 5 will come from vcid 30, and line #5 says frames 6 and 7 will come from vcid 18. The remaining "recur" lines define vcids for the remaining frames in the scenario. There is another way to specify units other than recurrent patterns. This is by unit range; see a packet mux for an example.

The second line defines an idle frame. (There are other idle definitions in the list.) It says the first frame is idle, and so is 101, 201, 301, etc. The idle argument, which is zero, is the bit length of the idle frame. Since all idle frames are the same fixed size, Tiger ignores the value, but it must exist.

The first line has three fields. "er(1)" turns on expected results output. The expected results file will contain unit ids (frame numbers) and corresponding vcids for all 1,000 frames. By default, this is off.

"default(vc17)" tells the mux to use input stream "vc17" if a unit id is undefined. For example, the mux would use the default if we forgot to identify an input stream for the 100th frame. It would also use the default if an input stream ran out of frames. For example, if we programmed "vc30" to only make 10 frames, the mux would get frames from the default for "vc30" after it used up the 10 frames. If we did not specify a default or if the default itself ran out of units, then the mux would terminate upon its next attempt to get a unit from the default.

"idle(vc17)" tells the mux from which stream to get idle units. Most of the time it does not matter which stream provides because they all can make idles. However, if we chose a CADU stream that

used an insert zone, then all idles from that stream would also have an insert zone. For our example, no CADU stream is using the insert zone, so any CADU stream will do.

```
stream.mux.top idle(vc17) er(1) default(vc17)
stream.mux.top.recur idle(0) start(1) repeat(0) span(100)
stream.mux.top.recur stream(vc17) start(2) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(4) repeat(1) span(100)
stream.mux.top.recur stream(vc18) start(6) repeat(1) span(100)
stream.mux.top.recur stream(vc42) start(8) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(10) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(12) repeat(1) span(100)
stream.mux.top.recur stream(vc41) start(14) repeat(0) span(100)
stream.mux.top.recur stream(vc18) start(15) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(17) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(19) repeat(1) span(100)
stream.mux.top.recur stream(vc41) start(21) repeat(1) span(100)
stream.mux.top.recur stream(vc23) start(23) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(25) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(27) repeat(1) span(100)
stream.mux.top.recur stream(vc18) start(29) repeat(1) span(100)
stream.mux.top.recur stream(vc42) start(31) repeat(1) span(100)
stream.mux.top.recur stream(vc41) start(33) repeat(0) span(100)
stream.mux.top.recur stream(vc30) start(34) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(36) repeat(1) span(100)
stream.mux.top.recur stream(vc23) start(38) repeat(1) span(100)
stream.mux.top.recur stream(vc18) start(40) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(42) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(44) repeat(1) span(100)
stream.mux.top.recur stream(vc41) start(46) repeat(1) span(100)
stream.mux.top.recur stream(vc23) start(48) repeat(0) span(100)
stream.mux.top.recur stream(vc42) start(49) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(51) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(53) repeat(1) span(100)
stream.mux.top.recur stream(vc18) start(55) repeat(1) span(100)
stream.mux.top.recur stream(vc23) start(57) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(59) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(61) repeat(1) span(100)
stream.mux.top.recur stream(vc11) start(63) repeat(0) span(100)
stream.mux.top.recur idle(0) start(64) repeat(1) span(100)
stream.mux.top.recur stream(vc11) start(65) repeat(1) span(100)
stream.mux.top.recur stream(vc18) start(66) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(68) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(70) repeat(1) span(100)
stream.mux.top.recur stream(vc42) start(72) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(74) repeat(3) span(100)
stream.mux.top.recur stream(vc17) start(78) repeat(1) span(100)
stream.mux.top.recur stream(vc18) start(80) repeat(1) span(100)
stream.mux.top.recur stream(vc42) start(82) repeat(1) span(100)
stream.mux.top.recur idle(0) start(84) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(85) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(87) repeat(1) span(100)
stream.mux.top.recur stream(vc41) start(89) repeat(0) span(100)
stream.mux.top.recur stream(vc18) start(90) repeat(2) span(100)
stream.mux.top.recur stream(vc30) start(93) repeat(1) span(100)
stream.mux.top.recur stream(vc17) start(95) repeat(1) span(100)
stream.mux.top.recur stream(vc30) start(97) repeat(0) span(100)
stream.mux.top.recur stream(vc2) start(98) repeat(2) span(100)
```

This stream makes CADUs for virtual channel 17. There are similar streams for the other virtual channels below. We have set it up to do path service. The spacecraft id is 42. We are Reed-Solomon encoding the frames with interleave 4 and codeword length 255. (If we had virtual fill, we would shorten the codeword length.) Notice that we do not specify the CADU length because the length is implied by the Reed-Solomon encoding. The length is interleave * codeword length + sync length, which is 1024 is this example.

The second "vc17" line tells the stream how to fill its data region. If we had an insert zone or an operational control field (OCF or CLCW), we would need similar lines for those regions. We are filling the "vc17" data region using type C, which is the consumer loader type. It fills the region with units from another stream. In this case, we fill with packets from the "pmux17" input stream, which is a mux for three packet streams. The loader will load packets back-to-back, splitting them between CADUs if necessary.

If the input stream runs out of packets, the consumer loader, by default, will put a constant fill pattern of 0xC9 into the data region to fill any remaining unused bytes. This is not the way we want to fill out any CADU. Instead, we would prefer to insert idle packets. The field "lastUnit(I)" does just that. If the input stream runs out of packets, the consumer loader will request idle packets. This will only happen in the last CADU produced by this stream. The loader and the stream will terminate after providing this lastCADU.

The "ERcomposition(1)" field is an optional field that causes the consumer loader to write unit composition information to the expected results file. By default it is off because it can produce lots of output. When on, the expected results file will show each CADU and the identity of all packets in it. It will also show the first header pointer value and the number of bytes per packet fragment when a CADU contains part of a split packet.

```
stream.cadu.vc17 service(P) vcid(17) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc17.region.data type(C) ERcomposition(1) inStream(pmux17) - lastUnit(I)
```

This mux stream interleaves the packets from three packet streams for input to "vc17." Notice that this mux uses a "range" specification. In this example, this means that the first ten packets (1-10) come from "ap256." In addition, packet 500 will also come from "ap256." The other streams use a recurrent pattern. The mux will get packets from "ap257" for packets 9-16, 19-26, 29-36, and so on. It will get packets from "ap258" for packets 17-18, 27-28, 37-38, and so on. Notice that some unit ids, such as #9, are in more than one list.

To resolve conflicts, a mux searches through the list in a specific order and uses the stream from the first match that it finds. A mux always searches through range definitions before it searches through any recurrent ones. To avoid confusion, we list ranges before recurs. Within ranges and recurs, a mux searches them in the order that they are listed in the script. If it still has not found a match, it uses the default, and if that fails (it does not exist or it ran out of units), the mux terminates. In the example, this mux gets packets 9, 10, and 500 from "ap256" because they are range lists, and it uses ranges over recurs.

```
stream.mux.pmux17 default(ap256) idle(ap256)
stream.mux.pmux17.range stream(ap256) uid0(1) uid1(10)
stream.mux.pmux17.range stream(ap256) uid0(500) uid1(500)
stream.mux.pmux17.recur stream(ap257) start(9) repeat(7) skip(2)
stream.mux.pmux17.recur stream(ap258) start(17) repeat(2) skip(8)
```

This stream makes packets for application id 256. Each packet is 332 bytes long, which includes the primary header length. We have chosen to put a six byte secondary header in each packet, and we have put a checksum at the end of each packet. The checksum is the sum of all bytes in the packet moduloe 256. Each packet has a data region and a secondary header region, so we must include lines to tell the packet stream how to fill them.

We fill the data region with a constant pattern of 0x11. Type F is a constant pattern loader. For the secondary header region, we are using the CCSDS unsegmented timecode. The timecode will increment by the step as we progress from packet to packet.

stream.pkt.ap256 appid(256) length(332) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap256.region.data type(F) pattern(0x11)
stream.pkt.ap256.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

This stream makes packets for application id 257.

stream.pkt.ap257 appid(257) length(340) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap257.region.data type(F) pattern(0x22)
stream.pkt.ap257.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

This stream makes packets for application id 258. In this stream, we will make the 30th packet a 40-byte idle packet.

stream.pkt.ap258 appid(258) length(128) 2hdr(1) 2hdrLength(6) checksum(1) - idleEvent(ap258idle)
stream.pkt.ap258.region.data type(F) pattern(0x33)
stream.pkt.ap258.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
event.val.ap258idle.range uid0(30) uid1(30) v(40)

This stream makes CADUs for virtual channel 18, and its layout is almost identical to that of "vc17." In this example, however, we will demonstrate how to introduce errors at the frame and packet levels.

There are two error specifications in this CADU stream. In the first, we plan to invert bit #2 (a bit in the spacecraft id field) in the second CADU. In the second, we plan to invert five bits (bits 100-104) in CADUs 5, 105, 205, 305, and 405.

The "convey" field defines how and when the error is applied. When convey is on, the error is conveyed to every receiving stream in the pipeline. For example, if we put a conveyable error in a packet, then the receiving frame stream and any other receiver will behave as if the error were introduced at that level. The important point to note is that the stream applies the error after it has encoded the unit when convey is on. If we put a bit error in a packet and piped it in a frame stream that was CRC encoding frames, then the affected frame would have a CRC error because the stream applied the error after it encoded the frame. If we then piped the frame into a NASCOM block stream, that NASCOM block would also have a CRC error. In all cases, the streams apply the error AFTER encoding when convey is on.

When convey is off, the stream that applies the error hides it from all receiving streams. This means the error is applied before encoding takes place. You might think of the error as an instrument error as opposed to a transmission error. For example, if we put a non-conveyable error in a packet, then we will not see CRC or RS errors in any frames containing that packet.

In summary, when convey is on, all receiving streams reflect the error, and the error is applied AFTER encoding. When convey is off, only the original stream shows the error, and the error is applied BEFORE encoding.

In this example, the spacecraft id error in the second CADU is non-conveyable, which means it is applied before the Reed-Solomon encoding. The target processor should detect a spacecraft id error. The other errors to bits 100-104 in the other five frames are conveyable, so the errors are applied after Reed-Solomon encoding. This means that the target processor should see Reed-Solomon errors, and they are correctable. The target processor should correct the errors in the five frames. The label field in both errors is a information tag that the stream writes to the expected results file.

stream.cadu.vc18 service(P) vcid(18) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc18.region.data type(C) ERcomposition(1) inStream(pmux18) - lastUnit(I)
stream.cadu.vc18.error.flip convey(0) label(spidError) event(ev0) - startbit(2) bits(1)
event.unit.ev0.range uid0(2) uid1(2)

stream.cadu.vc18.error.flip convey(1) label(rsError) event(ev1) - startbit(100) bits(5)
event.unit.ev1.recur start(5) skip(100) occur(5)

stream.mux.pmux18 default(ap259) idle(ap256)
stream.mux.pmux18.recur stream(ap259) start(1) repeat(5) span(13)
stream.mux.pmux18.recur stream(ap260) start(7) repeat(5) span(13)
stream.mux.pmux18.recur stream(ap261) start(13) repeat(0) span(13)

stream.pkt.ap259 appid(259) length(784) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap259.region.data type(F) pattern(0x44)
stream.pkt.ap259.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

This packet stream makes packets for application id 260. In this example, we throw away (drop) the second packet, which should cause a gap in the target processor. The expected results file will also note a gap. Notice that we share event "ev0," which we defined above to introduce a CADU error. Events are shareable.

stream.pkt.ap260 appid(260) length(392) 2hdr(1) 2hdrLength(6) checksum(1) - drop(ev0)
stream.pkt.ap260.region.data type(F) pattern(0x55)
stream.pkt.ap260.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

This packet stream makes packets for application id 261. We plan to deposit the value 1 into the packet data length field for packets 10-12. The error is non-conveyable, which means it is hidden from the CADU. In general, you should not convey packet errors to a CADU stream that is Reed-Solomon encoding frames because the target processor will probably remove them when the Reed-Solomon decoder detects and fixes the errors. You then will get Reed-Solomon corrected frames instead of your planned errors.

stream.pkt.ap261 appid(261) length(128) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap261.region.data type(F) pattern(0x66)
stream.pkt.ap261.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap261.error.set convey(0) label(length) event(ap261e) - startbit(32) bits(16) v(1)
event.unit.ap261e.range uid0(10) uid1(12)

----

stream.cadu.vc23 service(P) vcid(23) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc23.region.data type(C) ERcomposition(1) inStream(pmux23) - lastUnit(I)

stream.mux.pmux23 default(ap262) idle(ap256)
stream.mux.pmux23.recur stream(ap262) start(1) skip(1)
stream.mux.pmux23.recur stream(ap263) start(2) skip(1)

stream.pkt.ap262 appid(262) length(964) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap262.region.data type(F) pattern(0x77)
stream.pkt.ap262.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

This packet stream creates packets for application id 263. We will limit it to creating only 100 packets. The mux will use the default after this stream provides 100 packets.

stream.pkt.ap263 appid(263) length(944) 2hdr(1) 2hdrLength(6) checksum(1) - max(100)
stream.pkt.ap263.region.data type(F) pattern(0x88)
stream.pkt.ap263.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

----
stream.cadu.vc30 service(P) vcid(30) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc30.region.data type(C) ERcomposition(1) inStream(pmux30) - lastUnit(I)

stream.mux.pmux30 default(ap265) idle(ap256)
stream.mux.pmux30.recur stream(ap265) start(1) repeat(3) span(17)
stream.mux.pmux30.recur stream(ap266) start(5) repeat(3) span(17)
stream.mux.pmux30.recur stream(ap265) start(9) repeat(3) span(17)
stream.mux.pmux30.recur stream(ap266) start(13) repeat(1) span(17)
stream.mux.pmux30.recur stream(ap267) start(15) repeat(0) span(17)
stream.mux.pmux30.recur stream(ap266) start(16) repeat(1) span(17)

stream.pkt.ap265 appid(265) length(332) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap265.region.data type(F) pattern(0x99)
stream.pkt.ap265.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap266 appid(266) length(340) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap266.region.data type(F) pattern(0xaa)
stream.pkt.ap266.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap267 appid(267) length(128) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap267.region.data type(F) pattern(0xbb)
stream.pkt.ap267.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
----
stream.cadu.vc41 service(P) vcid(41) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc41.region.data type(C) ERcomposition(1) inStream(pmux41) - lastUnit(I)

stream.mux.pmux41 default(ap320) idle(ap256)
stream.mux.pmux41.recur stream(ap320) start(1) repeat(0) skip(4)
stream.mux.pmux41.recur stream(ap321) start(2) repeat(0) skip(4)
stream.mux.pmux41.recur stream(ap322) start(3) repeat(0) skip(4)
stream.mux.pmux41.recur stream(ap323) start(4) repeat(0) skip(4)
stream.mux.pmux41.recur stream(ap324) start(5) repeat(0) skip(4)

stream.pkt.ap320 appid(320) length(780) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap320.region.data type(F) pattern(0xcc)
stream.pkt.ap320.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap321 appid(321) length(580) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap321.region.data type(F) pattern(0xdd)
stream.pkt.ap321.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap322 appid(322) length(560) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap322.region.data type(F) pattern(0xee)
stream.pkt.ap322.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap323 appid(323) length(572) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap323.region.data type(F) pattern(0xff)
stream.pkt.ap323.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap324 appid(324) length(571) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap324.region.data type(F) pattern(0xa5)
stream.pkt.ap324.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
----

stream.cadu.vc42 service(P) vcid(42) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc42.region.data type(C) ERcomposition(1) inStream(pmux42) - lastUnit(I)

stream.mux.pmux42 default(ap67) idle(ap256)
stream.mux.pmux42.recur stream(ap67) start(1) repeat(2) skip(4)
stream.mux.pmux42.recur stream(ap68) start(4) repeat(2) skip(4)
stream.mux.pmux42.recur stream(ap69) start(7) repeat(0) skip(6)

stream.pkt.ap67 appid(67) length(642) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap67.region.data type(F) pattern(0x67)
stream.pkt.ap67.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap68 appid(68) length(642) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap68.region.data type(F) pattern(0x68)
stream.pkt.ap68.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
stream.pkt.ap69 appid(69) length(276) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap69.region.data type(F) pattern(0x69)
stream.pkt.ap69.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)
----
stream.cadu.vc11 service(P) vcid(11) spid(42) frameSync(0x1acffc1d) - RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc11.region.data type(C) ERcomposition(1) inStream(ap11) - lastUnit(I)

stream.pkt.ap11 appid(11) length(64) 2hdr(1) 2hdrLength(6) checksum(1)
stream.pkt.ap11.region.data type(F) pattern(0x88)
stream.pkt.ap11.region.2hdr type(tccuc) stepSeconds(0) coarseBytes(2) - fineBytes(3) stepFine(200)

This stream makes CADUs using a frame service (VCA/VCDU) for virtual channel 2. This time we have selected to use a twelve byte insert zone in every CADU. We will fill the data region with a random pattern. We will fill the insert zone with the user pattern 0x00faf30502ffd30403fc1303. However, we will put an 8-bit sequence number in the first byte of the insert zone.

stream.cadu.vc2 service(V) vcid(2) spid(42) frameSync(0x1acffc1d) - insertZone(12) RSencode(1) RSinterleave(4) RScodeLength(255)
stream.cadu.vc2.region.data type(A)
stream.cadu.vc2.region.insertZone type(U) wrap(0) format(H)
stream.cadu.vc2.region.insertZone.data s0(0x00) s1(0xfa) s2(0xf3) s3(0x05)
stream.cadu.vc2.region.insertZone.data s0(0x02) s1(0xff) s2(0xd3) s3(0x04)
stream.cadu.vc2.region.insertZone.data s0(0x03) s1(0xfc) s2(0x13) s3(0x03)
stream.cadu.vc2.task.sequence event(1) startbit(80) bits(8)

APPENDIX A
ERROR LISTING

Error #	Description
1	unsupported byte boundary
2	undefined name
3	Invalid path.
4	not a fancy input unit file invalid fancy file version number
5	invalid unit/record length not 4-byte divisible spacesize is not a multiple of recordsize
6	unknown event
7	missing path
8	missing field
9	invalid choice invalid value
10	open file failure
11	out of range
12	invalid region type
13	inStream provided no units
14	inStream could not provide idle unit
15	duplicate name
16	undefined stream
17	Resource is locked. Already used.
18	Input unit is bigger than the target region.
19	This stream cannot make idle units.
20	An input unit does not perfectly fit a region.
21	Mux needs idle stream definition.
22	Mux has no input streams.
23	You cannot position (seek) a fancy unit file.
24	length() must be specified and non-zero when only CRC encoding
25	Device must be file type.
26	region cannot be defined as EMPTY.
27	Event is inactive. No ranges or recurrent patterns exist.
28	Need event() or value() field.
29	A generic frame must have a sync pattern.
30	Frame is too short.
31	Region does not fit inside unit.
32	Record stream has no regions.
33	Cannot position input file for variable size units.
34	Failed to write fancy file header.
35	Failed to write TDG file header.
36	Failed to write TDG file trailer.
37	Failed to write STGEN file trailer.
38	Sim file output failure.
39	Stream is in an infinite loop.

100	v2PktStream	lsegment must be 256, 512, or 1024
101	v2PktStream	Received too-big idle packet from v1 packet stream
102	v1PktGroup	Group has only one packet
103	v1PktStream	A packet length is too small.
104	v1tfStream	lsegment must be 0, 256, 512, or 1024
105	tctfStream	Input unit is too big.
106	v1PktStream	Variable length packets specified but no lengths provided
107	v1tfStream	The V1TF length is too small.
108	caduStream	The CADU length is too short.
109	frames	The sync pattern length must be 0 or 4.
110	MasterChannel	Master channel got wrong size input frame.
111	tcSegStream	A MAP value is >= 64.
112	v1PktStream	A packet length is > 65,524.
113	tctfStream	Invalid bypass/control flags detected. Must be 0, 2, or 4.
114	tcPhysical	Both the idle and acquisition sequence lengths must be byte divisible.
115	tcPhysical	Acquisition sequence length is one bit! Not using value event?

Note: A message indicating "Tiger script error" is received instead of "SCTGEN script error" on the SCTGEN GUI main menu panel if an error occurs during SCTGEN script processing.